# Notices

**First Edition (June 1994)**

# Copyright Notices

# Disclaimers

# Trademarks

The following terms found in this publication, are trademarks of the IBM Corporation in the United States or other countries:

- OS/2
- DB2/2
- CICS
- IBM
- NetView
- Presentation Manager

The following terms found in this publication, are trademarks of other companies as follows. Other trademarks are trademarks of their respective companies.

| | |
|---|---|
| Intel | Intel Corporation |
| Lotus Notes | Lotus Development Corporation |
| Novell | Novell, Inc. |
| Compaq | Compaq Computer Corporation |
| Windows NT | Microsoft Corporation |
| Windows | Microsoft Corporation |
| ALR | Advanced Logic Research, Inc. |

# Overview of OS/2 for SMP Version 2.11

This document provides a guide for developers writing applications and device drivers for OS/2 for Symmetrical Multiprocessing (SMP) V2.11.

OS/2 for SMP V2.11 was developed to satisfy the need to run OS/2 on multiprocessor based CISC processors, namely the Intel x86 compatible family. The requirements for OS/2 for SMP V2.11 were that it run all existing applications, device drivers and subsystems, as well as take advantage of new multiprocessor (MP) exploitive applications and device drivers.

The emergence of low-cost MP hardware based on the 486 and Pentium processors makes OS/2 for SMP V2.11 an attractive desktop operating environment. Server and workstation environments using the x86 architecture are moving toward the more powerful emerging RISC based chip sets. These new RISC processors lack the full range of programming tools available for the x86 chip set. OS/2 for SMP V2.11 attempts to solve the problems of insufficient processor bandwidth by supporting multiple x86 processors in a single computer.

To provide increased performance, OS/2 for SMP V2.11 allows applications, file system, mass storage and network drivers to execute on any processor at any time. A number of databases and applications have been converted to run OS/2 for SMP V2.11. DB2/2 and CICS are two databases that IBM has converted to run under OS/2 for SMP V2.11. These application can benefit greatly from OS/2 for SMP V2.11 because they are CPU-intensive. Other applications which can also benefit from OS/2 for SMP V2.11 are:

- Lotus Notes & cc-Mail
- NetView Series
- Novell Netware
- Scientific Applications
- MultiMedia Applications
- OS/2 Lan Server

# Architectural Design Objectives

The architectural design objectives for a multiprocessor (MP) version of OS/2 were as follows:

1. Transparent support for two or more CPUs (16 CPUs max).
2. Support for applications and device drivers which are not MP safe and aware.
3. Support for various MP hardware platforms (eg. Compaq, APIC, EBI2, Corollary, etc) via a Platform Specific Driver (PSD) layer.
4. Small footprint - 4MB for OS/2 and DOS applications; 6MB for WINOS2
5. Performance equal to or better than Windows NT.
6. 100% application compatible for existing OS/2 DOS and Windows applications.
7. Honor priority preemption across all processors.

Transforming the OS/2 2.x uniprocessor (UP) code base into OS/2 for SMP V2.11 was mostly a matter of copying the vital system data structures for the number of processors. Only ONE copy of OS/2 is running at one time, no matter how many processors are present. System initialization automatically determines the number of processors and generates the appropriate number of data structures, including new control blocks and per-processor data structures. one process/thread running

# Platform Specific Drivers (PSDs)

In OS/2 for SMP V2.11, all of the platform specific code has been removed from the operating system, and placed into a Platform Specific Driver. These drivers provide an abstraction layer for the underlying hardware by allowing the operating system to call generic functions to perform platform-specific operations without worrying about the actual hardware implementation. This allows OS/2 for SMP V2.11 to support new MP hardware platforms without modifying the operating system.

PSDs are 32-bit flat DLLs specified in CONFIG.SYS by using the PSD= keyword, and must conform to the 8.3 file naming convention (e.g. PSD=BELIZE.PSD). They cannot contain either drive or path information because OS/2 cannot process such information at the stage of the startup sequence when the PSD statements are processed. The root directory of the startup partition is first searched for the specified file name, followed by the \OS2 directory of the startup partition. If drive or path information is included in a PSD statement, an error is generated.

PSD parameters may be specified after the PSD's name, and may be a maximum of 1024 characters long. The parameter string is not interpreted or parsed by OS/2, but is passed verbatim as an ASCIIZ string when the PSD's Install function is invoked.

If multiple PSD statements are encountered, OS/2 will load each PSD in the order listed in CONFIG.SYS, and call the PSD's install function. The first PSD which successfully installs will be the one OS/2 uses.

PSD statements are processed before BASEDEV, IFS, and DEVICE statements.

# Platform Specific Driver Architecture and Structure

The PSD operates in three contexts (modes): Kernel, Interrupt, and Init.

- **Kernel Mode**

  The OS/2 kernel calls the PSD for task-time operations, that is, it will execute as a thread within a process. Kernel mode is also referred to as the task context.

- **Interrupt Mode**

  The OS/2 kernel calls the PSD for interrupt-time operations. Interrupt time is a generic term that refers to executing code as a result of a hardware interrupt. The code does not execute as a thread belonging to a process.

- **Init Mode**

  The PSD is currently being used for system initialization. A limited set of PSD helps are available for use.

PSDs may contain multiple code and data objects. All objects will be fixed (not-swappable or movable) in low physical memory, with virtual addresses in the system arena. Objects are loaded in low physical memory to facilitate the use of real mode or bi-modal code. All objects default to permanent, which means that they remain in the system after initialization is completed. The SEGMENTS directive and the IOPL option in the linker DEF file should be used to mark those objects that are not to be kept after initialization.

The multitasking/multiprocessing environment of OS/2 for SMP V2.11 dictates that the PSD must be capable of handling multiple requests simultaneously. This means that global variables should be used sparingly. Upon PSD installation, the kernel passes a pointer to a small area of processor local memory (PLMA) which the PSD developer can use to store variables. PSD developers must be aware of the effects of the calls they make, because there is no guarantee that if an operation is started on a processor, and execution blocks, that the operation will continue on the same processor. OS/2 does not preempt a thread in the PSD, but it may block as a result of using a PSD help, or it may be interrupted by a hardware interrupt.

PSDs can register an interrupt handler for any given IRQ level using the SET_IRQ PSD help. These interrupt handlers are guaranteed to be called before any device driver's interrupt handler. If the PSD's interrupt handler returns NO_ERROR, the interrupt manager will assume the interrupt has been handled, it will end the interrupt. If a -1 is returned, the interrupt manager will assume that the interrupt has not been handled, and will call each device driver which has a registered interrupt handler for that particular level until one claims the interrupt. If the interrupt is unclaimed, the IRQ level will be masked off.

All PSDs must use the SET_IRQ PSD help to indicate which IRQ level they will be

using for inter-processor interrupts (IPI). If the PSD's IPI IRQ level is shared, it must register a handler which detects if the IRQ is an IPI or another interrupt. The handler must return NO_ERROR if the interrupt was caused by an IPI, otherwise it returns a -1. If the IPI IRQ level is unique, an interrupt handler need not be installed but SET_IRQ must still be used to indicate which is the IPI IRQ level.

The kernel will save the state of all the registers (except EAX) around calls to the PSD functions. All the functions will run at Ring 0. Upon invocation, SS, DS, and ES will be flat. The PSD functions must conform to the C calling convention. They receive parameters on the stack (4 bytes per parameter), and must return a return code in EAX.

The PSD functions have been split into three categories:

- o Functions that the PSD must have for OS/2 to operate (required functions)
- o Functions that the PSD does not need to have (optional functions)
- o Functions that the PSD must have for OS/2 to use multiple processors (MP functions).

The kernel provides default handling for some of the PSD functions. PSD functions can also chain to a kernel default handler by returning a -1 return code. If a return code other than -1 is returned by a PSD function, the default handler will not get called. The PSD function glossary later in this chapter details the categories of all the functions, as well as any default handlers they may have.

The PSD developer makes functions available to OS/2 by using the EXPORTS statement in the module definition (DEF) file. All functions should be exported using upper case with no leading underscores (_). An example is shown below.

```
LIBRARY TESTPSD

EXPORTS

    PSD_INSTALL   = _Install
    PSD_DEINSTALL = _DeInstall
```

The initial CS and EIP in the PSD's executable image is ignored. The image should also not contain a stack object. OS/2 allocates a per-processor PSD stack and sets SS and ESP correctly before invoking any of the PSD functions.

PSDs should be written in flat 32-bit code, using C, and must be linked as a LIBRARY.

OS/2 invokes all PSD functions in protect mode, but there is also a PSD help which allows the PSD developer to call a PSD function in real mode.

OS/2 services are provided through the PSD help interface. Access to these services are obtained upon PSD installation. PSD helpers preserve all registers

except EAX.

All the definitions (e.g. defines, structures, etc.) that are required for building a PSD are in the header file PSD.H.

# OS/2 Initialization

OS/2 requires a PSD for system initialization. The system will display an error message if a valid PSD for the current platform cannot be installed.

The following is a list of steps, in the order in which they occur, that are executed after a PSD is installed. If any step does not complete successfully, the system initialization process will stop, and a fatal error message will be displayed.

1. After a PSD is successfully installed, its Init function is invoked. This function is used to allocate and initialize any resources that the PSD may require, as well as initializing the state of the hardware.

2. The kernel determines the number of usable processors on the current platform by using the PSD_GET_NUM_OF_PROCS function.

3. The kernel allocates all resources required to support the additional processors. This step determines what to allocate based on the results of the previous step.

4. The PSD's processor initialization function is invoked on the current processor (CPU0).

5. An MP daemon is created for CPU0. An MP daemon is a thread that never goes away, which is used for MP operations by a specific processor.

6. An MP daemon is created for the next logical processor.

7. The PSD's start processor call is invoked to start the next logical processor. The PSD should only start the specified processor, and then return (see the PSD_START_PROC function for more detail). The started processor will spin in a tight loop waiting for a variable to be cleared. This variable is referred to as the processor initialization real mode spinlock.

8. Upon return from the PSD's start processor call, the processor initialization real mode spinlock is cleared.

9. CPU0 will spin in a tight loop waiting for a variable to be cleared. This variable is referred to as the CPU0 spinlock.

10. The started processor continues execution of the kernel's real mode processor initialization code now that processor's initialization real mode spinlock has been cleared.

11. The started processor sets up all protect mode and paging information, and switches into protect mode with paging enabled.

12. Up to this point, the started processor has been running on a small

processor initialization stack (It has not been running as an OS/2 thread). The current context is switched to that of this processors MP daemon.

13. OS/2 calls the PSD's processor initialization function for the current processor.

14. The PSD indicates that the processor has been initialized.

15. The started processor will spin in a tight loop waiting for a variable to be cleared. This variable is referred to as the processor initialization protect mode spinlock.

16. The CPU0 spinlock is cleared.

17. System initialization continues on CPU0 now that its spinlock has been cleared.

18. Steps 6, through 17 are repeated until all processors have been started.

19. The rest of system initialization continues normally, on CPU0.

20. After the system is fully initialized, the processor initialization protect mode spinlock is cleared. This allows CPU1 through CPU-N to start executing code.

# PSD Function Glossary

In the functions listed below all pointers must be flat 32-bit linear addresses.

The following keywords indicate:

| | |
|---|---|
| **Required** | Indicates that the function is required for OS/2 to operate properly, so the function can not be omitted. |
| **Optional** | Indicates that the function is not required. |
| **MP** | Indicates that the function is not required for OS/2 to execute with one processor, but it is required for OS/2 to use multiple processors. |
| **Default** | Indicates that the OS/2 kernel provides default handling for that specific function. |
| **Can Block** | Indicates that the function can call a PSD help that may block. |
| **Can't Block** | Indicates that the function can not call a PSD help that may block. |
| **Input** | Indicates that the kernel fills the field with input values before calling the function. |
| **Output** | Indicates that the PSD should return values in the specified field. |
| **0-based** | Indicates that a zero denotes the first value. |
| **1-based** | Indicates that a one denotes the first value. |

# PSD Functions

Following are the PSD functions.

# PSD_INSTALL

**PSD_INSTALL keywords**

Required, Can Block

**Description**

Determine if the PSD supports the current platform.

This function probes the hardware to see if the PSD supports the current platform. No other operations should be executed in this function. It is merely a presence check. This function is the first function called upon loading a PSD. It must store away all the information passed to it in the install structure.

**Mode**

Called in Init Mode; may be called in Kernel Mode.

**Entry**

Pointer to an INSTALL structure.

**Exit**

| | |
|---|---|
| NO_ERROR | if the PSD installed successfully. |
| -1 | if the PSD does not support the current platform. |

**Structures**

```
typedef struct install_s
{
   P_F_2   pPSDHlpRouter;  (Input)
   char   *pParmString;    (Input)
   void   *pPSDPLMA;       (Input)
   ulong_t sizePLMA;       (Input)
} INSTALL;
```

| | |
|---|---|
| pPSDHlpRouter | points to the PSD help router. Use the PSDHelp macro in PSD.H to access the PSD helps. |
| pParmString | points to any parameters specified in CONFIG.SYS after the PSD's name. If no parameters were specified this field is NULL. |
| pPSDPLMA | points to the PSD's processor local memory area. This area contains different physical memory at the same linear address across all processors. You can use the |

area to store variables such that each processor accesses unique values, but all the code references the same variables.

sizePLMA — is the total size of the PSD's PLMA in bytes.

**Notes**

This function may be called after OS/2 is finished with initialization by the Dos32TestPSD API; therefore, the PSD developer must be careful not to use any Init mode only PSD help's in this function.

# PSD_DEINSTALL

**PSD_DEINSTALL keywords**

Required, Can Block

**Description**

DeInstall the PSD.

This function is called to release any resources that may have been allocated by the PSD_INSTALL function. A PSD is never de-installed after its Init routine is called.

**Mode**

Called in Init Mode; may be called in Kernel Mode.

**Entry**

None.

**Exit**

| | |
|---|---|
| NO_ERROR | if the PSD DeInstalled succesfully. |
| -1 | if the PSD didn't DeInstall. |

**Notes**

This function may be called after OS/2 is finished with initialization by the Dos32TestPSD API; therefore, the PSD developer must be careful not to use any init mode only PSDHelp's in this function.

# PSD_INIT

**PSD_INIT keywords**

Required, Can Block

**Description**

Initialize the PSD.

This function is called to initialize the PSD. It is used to allocate and initialize any resources that the PSD may require, as well as initializing the state of the hardware. This function should only initialize the state of the hardware in general. Initialization of CPUs should be done in ProcInit. It must fill in the INIT structure passed to it by OS/2. This function is only called once on CPU0.

**Mode**

Called in Init Mode only.

**Entry**

Pointer to INIT structure

**Exit**

| | |
|---|---|
| NO_ERROR | if the PSD initialized successfully. |
| -1 | if the PSD didn't initialize. |

**Structures**

```
typedef struct init_s
{
   ulong_t flags;     (Output)
   ulong_t version;   (Output)
} INIT;
```

flags in the INIT structure indicate any special features or requirement that the PSD may have.

INIT_GLOBAL_IRQ_ACCESS indicates that the platform can perform IRQ operations (e.g. PIC masking) on any processor. If this flag is omitted, the IRQ functions are guaranteed to only get called on CPU0, otherwise they may get called

| | |
|---|---|
| | on any processor. If the flag is omitted and an IRQ operation is initiated on a processor other then CPU0, the OS/2 kernel will route the request to CPU0. |
| INIT_USE_FPERR_TRAP | indicates that Trap 16 will be used to report floating point errors, instead of IRQ 13. If this flag is set, the kernel sets the NE flag in CR0 for all processors. The PSD is responsible for doing any additional work for making the transition. |
| INIT_EOI_IRQ13_ON_CPU0 | indicates that an EOI for a floating point error using IRQ13 should only be performed from CPU0. On CPU1-N, the hardware is responsible for clearing the interrupt. |

version indicates the version number of this PSD. It should be updated appropriately as this will help with service.

**Notes**

None.

# PSD_PROC_INIT

**PSD_PROC_INIT keywords**

MP, Can Block

**Description**

Initialize the current processor.

This function is called to initialize the current processor. It is called in protect mode, once on a per-processor basis. It should initialize variables in the PSD's PLMA, along with initialization of the hardware state for that specific processor.

**Mode**

Called in Init Mode only.

**Entry**

None.

**Exit**

| | |
|---|---|
| NO_ERROR | if the processor initialized successfully. |
| -1 | if the processor didn't initialize. |

**Structures**

None

**Notes**

None

# PSD_START_PROC

**PSD_START_PROC keywords**

MP, Can Block

**Description**

Start a processor.

This function is used to start a specified processor. The PSD may only start the processor that was specified.

OS/2 fills in the address of a started processors initial real mode CS:IP in the warm reboot vector of the BIOS data area (0x40:0x67).

OS/2 provides serialization such that another processor will not be started until the previous processor has finished its real mode initialization, gone into protect mode, and finished calling the ProcInit function. The processor which is started will be held in real mode until the StartProc function has been completed, and will then be allowed to initialize.

All processors are started before the first device driver is loaded.

**Mode**

Called in Init Mode only.

**Entry**

Processor number (0-based).

**Exit**

| | |
|---|---|
| NO_ERROR | if the processor started successfully. |
| -1 | if the processor didn't start. |

**Structures**

None.

**Notes**

If the hardware implementation uses some other mechanism to indicate a started processors initial CS:IP the value specified in the warm reboot vector should be used.

If the hardware implementation requires some other real mode operation to be completed before the processor can continue to execute, the PSD developer must be certain to chain to the address specified in the warm reboot vector.

# PSD_GET_NUM_OF_PROCS

**PSD_GET_NUM_OF_PROCS keywords**

Required, Can Block

**Description**

Return number of processors.

This function must detect and return the number of usable x86 based processors that exist on the current platform. If the PSD detects that any of the processors are defective or non x86-based, it is the PSD's responsibility to setup the state of the PSD and hardware, such that all usable processors are logically ordered. For example, if there are 4 processors and CPU2 is defective, the CPU's should be ordered as follows: CPU0 = 0, CPU1 = 1, CPU2 (Defective), CPU3 = 2).

**Mode**

Called in Init Mode only.

**Entry**

None.

**Exit**

Number of processors (1-based).

**Structures**

None.

**Notes**

OS/2 for SMP V2.11 only supports processors that are compatible with the architecture of the Intel 386 and above.

# PSD_GEN_IPI

**PSD_GEN_IPI keywords**

MP, Can't Block

**Description**

Generate an inter-processor interrupt.

This function is used to generate an inter-processor interrupt. All inter-processor hardware dependencies should be fully initialized before the first GenIPI is called.

**Mode**

Called in Kernel, and Interrupt Mode.

**Entry**

Processor number to interrupt (0-based).

**Exit**

| | |
|---|---|
| NO_ERROR | if the IPI was generated. |
| -1 | if the IPI was not generated. |

**Structures**

None.

**Notes**

OS/2 guarantees that the GenIPI function will not be called to interrupt a processor that has not finished processing any previous IPIs.

# PSD_END_IPI

**PSD_END_IPI keywords**

MP, Can't Block

**Description**

End an inter-processor interrupt.

This function is used to end an inter-processor interrupt, that was generated by GenIPI.

**Mode**

Called in Kernel, and Interrupt Mode.

**Entry**

Processor number to end interrupt on (0-based).

**Exit**

| | |
|---|---|
| NO_ERROR | if the IPI was ended successfully. |
| -1 | if the IPI didn't end successfully. |

**Structures**

None.

**Notes**

The processor number specified and the current processor number should be identical.

# PSD_PORT_IO

**PSD_PORT_IO keywords**

Optional, Default, Can't Block

**Description**

Perform local port I/O.

Some platforms have some non MP specific system ports localized on a per-processor basis. If a local I/O operation may block before completion, I/O can be routed to a specific CPU for processing. This should be done, because an operation which started on one processor is not guaranteed to complete on that processor if execution is blocked. This function gets invoked as the result of a device driver calling DevHelp_Port_IO.

**Mode**

Called in Kernel, and Interrupt Mode.

**Entry**

Pointer to a PORT_IO structure.

**Exit**

| | |
|---|---|
| NO_ERROR | if the I/O was successful. |
| -1 | if the I/O wasn't successful. |

**Structures**

```
typedef struct port_io_s
{
   ulong_t port;    (Input)
   ulong_t data;    (Input/Output)
   ulong_t flags;   (Input)
} PORT_IO;
```

| | |
|---|---|
| port | indicates which port to read to, or write from. |
| data | contains the data read from a read request, or the data to write if a write request. If the request uses less the 4 bytes the least significant portion of the data variable is used. |
| flags | indicate what operation to perform. |

| | |
|---|---|
| IO_READ_BYTE | Read a byte from the port |
| IO_READ_WORD | Read a word from the port |

| | |
|---|---|
| IO_READ_DWORD | Read a dword from the port |
| IO_WRITE_BYTE | Write a byte to the port |
| IO_WRITE_WORD | Write a word to the port |
| IO_WRITE_DWORD | Write a dword to the port |

**Notes**

If the I/O performed is to a non-local port, the I/O should be handled as a regular I/O request.

If device drivers or applications access the local ports directly, instead of using the documented interfaces problems may occur.

# PSD_IRQ_MASK

**PSD_IRQ_MASK keywords**

Optional, Default, Can't Block

**Description**

Mask/Unmask IRQ levels

This function allows masking (disabling), or un-masking (enabling) of IRQ levels. When this function is invoked it should save the state of the interrupt flag, and disable interrupts before performing the mask operation. It should then restore the state of the interrupt flag.

**Mode**

Called in Kernel, and Interrupt Mode.

**Entry**

Pointer to PSD_IRQ structure.

**Exit**

| | |
|---|---|
| NO_ERROR | operation completed successfully. |
| -1 | operation failed. |

**Structures**

```
typedef struct psd_irq_s
{
   ulong_t flags;     (Input)
   ulong_t data;      (Input/Output)
   ulong_t procnum;   (Input)
} PSD_IRQ;
```

| | |
|---|---|
| data | is the logical IRQ levels to mask, or un-mask. |
| flags | indicate which type of operation is to be performed. |

| | |
|---|---|
| IRQ_MASK | mask (disable) IRQ levels |
| IRQ_UNMASK | unmask (enable) IRQ levels |
| IRQ_GETMASK | retrieves the masks for all IRQ levels |
| IRQ_NEWMASK | indicates that all the IRQ levels should reflect the state of the specified mask. |

procnum is the processor number of where the operation should take place.

**Notes**

If this function is omitted, OS/2 will perform all mask operations for an 8259 Master/Slave based PIC system. The requests will be sent to CPU0 depending on the state of the INIT_GLOBAL_IRQ_ACCESS flag.

# PSD_IRQ_REG

**PSD_IRQ_REG keywords**

Optional, Default, Can't Block

**Description**

Access IRQ related registers.

This function permits access to the IRQ related registers.

**Mode**

Called in Kernel, and Interrupt Mode.

**Entry**

Pointer to PSD_IRQ structure.

**Exit**

| | |
|---|---|
| NO_ERROR | operation completed successfully. |
| -1 | operation failed. |

**Structures**

```
typedef struct psd_irq_s
{
   ulong_t flags;     (Input)
   ulong_t data;      (Input/Output)
   ulong_t procnum;   (Input)
} PSD_IRQ;
```

flags — indicate which type of operation is to be performed.

| | |
|---|---|
| IRQ_READ_IRR | read the interrupt request register. |
| IRQ_READ_ISR | read the in service register. |

data — contains the data read from a read request, or the data to write if a write request.

procnum — is the processor number of where the operation should take place.

**Notes**

If this function is omitted, OS/2 will perform all register operations for

an 8259 Master/Slave based PIC system. The requests will be sent to CPU0 depending on the state of the INIT_GLOBAL_IRQ_ACCESS flag.

# PSD_IRQ_EOI

**PSD_IRQ_EOI keywords**
Optional, Default, Can't Block

**Description**
Issue an EOI.

This function is used to issue an End-Of-Interrupt.

**Mode**
Called in Kernel, and Interrupt mode.

**Entry**
Pointer to PSD_IRQ structure.

**Exit**

| | |
|---|---|
| NO_ERROR | operation completed successfully. |
| -1 | operation failed. |

**Structures**

```
typedef struct psd_irq_s
{
   ulong_t flags;     (Input)
   ulong_t data;      (Input/Output)
   ulong_t procnum;   (Input)
} PSD_IRQ;
```

| | |
|---|---|
| data | is the interrupt level to end. |
| flags | is not used in this operation. |
| procnum | is the processor number of where the operation should take place. |

**Notes**
If this function is omitted, OS/2 will perform all EOI operations for an 8259 Master/Slave based PIC system. The requests will be sent to CPU0 depending on the state of the INIT_GLOBAL_IRQ_ACCESS flag.

# PSD_APP_COMM

**PSD_APP_COMM keywords**

Optional, Can Block

**Description**

Perform generic APP/PSD communication.

This function performs generic application/PSD communication. The entry arguments, and return codes are not interpreted by OS/2, it is passed verbatim to and from the PSD.

**Mode**

Called in Kernel mode.

**Entry**

Function number, Argument.

**Exit**

Return code.

**Structures**

None.

**Notes**

None.

# PSD_SET_ADV_INT_MODE

**PSD_SET_ADV_INT_MODE keywords**

Optional, Can't Block

**Description**

TBD

**Mode**

Called in Init Mode only.

**Entry**

None.

**Exit**

Return code.

**Structures**

None.

**Notes**

The kernel initially provides default handling/detection for spurious interrupts. This is done for the last IRQ line of every PIC. It does this by checking the PIC's ISR register, and if the IRQ is not in service, it does not pass the interrupt request to the interrupt manager (i.e. a spurious interrupt).

If a PSD switches into advanced interrupt mode; the kernel will no longer provide default handling/detection of spurious interrupts. It becomes the PSD's responsibility.

One way a PSD could provide handling/detection of a spurious interrupt is to register a PSD handler for an IRQ level which may be spurious. As soon as the interrupt is detected the handler should insure that it is valid. If it is not (i.e. a spurious interrupt), it should dismiss the interrupt, and return NO_ERROR to the interrupt manager. The NO_ERROR return code informs the interrupt manager that the interrupt has been handled by the PSD. If the interrupt is valid the PSD should return a -1, as this informs the interrupt manager that the interrupt should be passed on to any device drivers registered to receive that interrupt.

# PSD Helps

OS/2 provides system services to the PSD developer via PSD helps.

The address of the PSD help router is passed in the INSTALL structure when a PSD's install function is called. All PSD helps destroy the contents of the EAX register (used for a return code). All other registers, including the flags, are preserved.

To invoke a PSD help, set up the appropriate parameters and call the PSD help router. For an example, refer to Appendix A. Some prototypes and macros are defined in PSD.H to simplify their usage.

The following keywords indicate:

| | |
|---|---|
| **May Block** | Indicates that the help may block. |
| **Won't Block** | Indicates that the help won't block. |

# PSDHLP_VMALLOC

**PSDHLP_VMALLOC keywords**

May Block

**Description**

Allocate memory.

This function is used to allocate virtual memory, or map virtual memory to physical memory, depending on the value of the flags. All virtual addresses are allocated from the system arena (i.e. global address space).

**Mode**

Callable in Init and Kernel mode.

**Parameters**

Pointer to a VMALLOC structure.

**Exit**

Return code.

**Structures**

```
typedef struct vmalloc_s
{
   ulong_t addr;     (Input/Output)
   ulong_t cbsize;   (Input)
   ulong_t flags;    (Input)
} VMALLOC;
```

addr — is filled with the linear address of the allocated or mapped memory on return from the help.

If VMALLOC_LOCSPECIFIC is specified, this field must contain the virtual address to map before calling the help.

If VMALLOC_PHYS is specified, this field must contain the physical address to map before calling the help.

cbsize — is the size of the allocation, or mapping in bytes.

flags — indicate which type of operation is to be performed.

VMALLOC_FIXED — indicates that the allocated memory is to be fixed in memory (not-swappable or movable). If

| | |
|---|---|
| | this flag is omitted, the allocated memory will be swappable by default. |
| VMALLOC_CONTIG | indicates that the allocation must use contigous physical memory. If this flag is specified VMALLOC_FIXED must also be used. |
| VMALLOC_LOCSPECIFIC | indicates a request for a memory allocation at a specific virtual address. If this flag is specified, the addr field must contain the virtual address to map.<br><br>**Note:** This flag can be used with the VMALLOC_PHYS flag to allocate memory where linear = physical. |
| VMALLOC_PHYS | indicates a request for a virtual mapping of physical memory. If this flag is specified, the addr field must contain the physical address to map.<br><br>**Note:** This flag can be used with the VMALLOC_LOCSPECIFIC flag to allocate memory where linear = physical. |
| VMALLOC_1M | indicates a request for a memory allocation below the 1MB boundary. |

**Notes**

None.

# PSDHLP_VMFREE

**PSDHLP_VMFREE keywords**

May Block

**Description**

Free allocation created by PSDHLP_VMALLOC.

This function frees memory or destroys a physical mapping created by the PSDHLP_VMALLOC help.

**Mode**

Callable in Init and Kernel Mode.

**Parameters**

Linear address to free.

**Exit**

Return code.

**Structures**

None.

**Notes**

All memory or mappings allocated by a PSD must be released if the PSD is DeInstalled.

# PSDHLP_SET_IRQ

**PSDHLP_SET_IRQ keywords**

Won't Block

**Description**

Setup IRQ information.

This function is used to setup IRQ information.

The PSD can use this help to register an interrupt handler at any given IRQ level between IRQ 0-IRQ 1F. These interrupt handler's are guaranteed to be called before any device driver's interrupt handler. If the PSD's interrupt handler returns NO_ERROR, the interrupt manager will assume the interrupt has been handled, and it will end the interrupt. If a -1 is returned, the interrupt manager will assume that the interrupt has not been handled, and will call each device driver which has a registered interrupt handler for that particular level, until one claims the interrupt. If the interrupt is unclaimed, the IRQ level will be masked off.

All PSDs must use the SET_IRQ PSD help to indicate which IRQ level it will be using for its inter-processor interrupts (IPI). If the PSD's IPI IRQ level is shared, it must register a handler which detects if the IRQ is an IPI or another interrupt. The handler must return NO_ERROR if the interrupt was caused by an IPI, otherwise, it returns a -1. If the IPI IRQ level is unique, an interrupt handler need not be installed, but SET_IRQ must still be used to indicate the IPI IRQ level.

This function can also be used to set, or remap what interrupt vector a particular IRQ level will use.

**Mode**

Callable in Init mode only.

**Parameters**

Pointer to a SET_IRQ structure.

**Exit**

Return code.

**Structures**

```
typedef struct set_irq_s
{
```

```
    ushort_t irq;
    ushort_t flags;
    ulong_t  vector;
    P_F_2    handler;
} SET_IRQ;
```

irq — specifies which IRQ level this operation is to be performed on.

flags — indicate what is the type of the specified IRQ. If no flag is used, a regular IRQ level is assumed.

- IRQf_IPI — indicates that the specified IRQ level is to be used for inter-processor interrupts.
- IRQf_LSI — indicates that the specified IRQ level is to be used as a local software interrupt.
- IRQf_SPI — indicates that the specfied IRQ level is to be used as a system priority interrupt.

vector — is used to specify what interrupt vector the IRQ level will use.

handler — contains the address of an interrupt handler. If the PSD is just specifying that a specific IRQ level is of a special type (e.g. IPI IRQ), it does not need a handler (the handler variable must be NULL).

**Notes**

IRQf_LSI, and IRQf_SPI, are currently not being used.

# PSDHLP_CALL_REAL_MODE

**PSDHLP_CALL_REAL_MODE keywords**

Won't Block

**Description**

Call a PSD function in real mode.

This function is used by the PSD developer to call one of his PSD functions in real mode.

**Mode**

Callable in Init mode only.

**Parameters**

Pointer to a CALL_REAL_MODE structure.

**Exit**

Called functions return code.

**Structures**

```
typedef struct call_real_mode_s
{
   ulong_t function;
   ulong_t pdata;
} CALL_REAL_MODE;
```

| | |
|---|---|
| function | contains the linear address of the function to be called in real mode. |
| pdata | contains the linear address of a parameter to be passed to the real mode function. The parameter is pointed to by DS:SI on entry to the called function. |

A return code may be returned by the real mode function in EAX.

**Notes**

No PSD helps may be used in real mode.

# PSDHLP_VMLINTOPHYS

**PSDHLP_VMLINTOPHYS keywords**

Won't Block

**Description**

Convert linear address to physical

This function converts the specified linear address to physical.

**Mode**

Callable in Init, Kernel, and Interrupt Mode.

**Parameters**

Linear address to convert.

**Exit**

Physical address if successful, -1 otherwise.

**Structures**

None.

**Notes**

None.

# Application Programming Interface

The OS/2 kernel will provide new support APIs for PSDs.

# DosCallPSD

**Description**

Perform generic APP/PSD communication.

This function performs generic application/PSD communication. The entry arguments, and return codes are not interpreted by OS/2, it is passed verbatim to, and from the PSD.

**Parameters**

Function number, Argument.

**Exit**

Return code.

**Notes**

This function can only be called from protect mode applications. There is currently no plan to provide a DOS mode equivalent.

If the PSD does not export the PSD_APP_COMM function, and the DosCallPSD API is invoked, ERROR_INVALID_FUNCTION is returned.

# Dos32TestPSD

**Description**

Determine if the PSD is valid for the current platform.

This function will load the specified PSD, call its Install, and DeInstall routine, and unload the PSD. It returns the return code from the PSD's Install routine, or any error code it might have received while attempting to do the above operation.

**Parameters**

Pointer to a fully qualified PSD path and name.

**Exit**

Return Code.

**Notes**

This function will mainly be used by OS/2's install, to test the validity of a PSD that will be installed.

# Understanding Spinlocks

OS/2 for SMP V2.11 provides synchronization and serialization using spinlocks. A spinlock is simply a section of code that executes in a tight loop waiting for a variable to be cleared.

# Spinlocks

The defined mechanism for protection of critical resources in OS/2 MP is a spinlock. A spinlock is a serialization mechanism that is used to restrict access to a critical resource to the owner of the spinlock. Spinlocks are implemented in the LockManager, which is part of the kernel, and are manipulated by using the new MPHelper services. The act of acquiring a spinlock can be thought of as locking the spinlock.

The reason spinlocks are used for critical resource protection instead of disabling interrupts, is disabling interrupts no longer works in an MP environment. CLI and STI will only work on the processor on which the instruction is executing. It will not prevent another processor from executing task time or interrupt code from the same device driver. Even though the kernel is non-preemptive, another processor can enter the kernel, and hence a device driver at any time. That means that CLI will not provide the same protection as on a single processor system. CLI/STI must be avoided.

MP aware device drivers will use spinlocks to protect critical resources. A spinlock must be allocated for each critical resource. Spinlock allocation should be done during initialization. When access to the resource is required, the device driver will try to lock the spinlock for the resource. If the spinlock is already locked then the processor will "spin" waiting for the lock to become available. Once the spinlock is acquired (locked) the device driver has exclusive access to the critical resource. The spinlock should remain locked for a VERY short time. When done with the resource the spinlock is released (unlocked).

Because spinlocks are for VERY SHORT durations, interrupts must be disabled while a spinlock is locked. If interrupts were enabled the path of execution could be expanded indefinitely by interrupt handlers. To enforce this rule, the LockManager will save the state of the interrupt flag and disable interrupts when a spinlock is locked. When the spinlock is unlocked, the LockManager will restore the interrupt flag to his original state. This allows device drivers to be unaware of the interrupt flag state while locking and unlocking spinlocks. The device driver, however, must not enable interrupts while owning a spinlock. If interrupts were enabled there are two possible effects. First is the uncontrolled expansion of the time a spinlock is owned. Second is the possibility of deadlock.

A spinlock is defined such that an attempt to acquire a spinlock which is currently owned by another processor, makes you spin (i.e. a tight loop of code which waits for the spinlock to be released). Spinlocks should be used sparingly, and should only be owned for very short periods of time, as spinning prevents the processor from doing any additional work. Spinlocks have different properties depending on whether it is a kernel or device driver spinlock. Spinlocks have been used because it is more expensive to reschedule a thread that is trying to acquire a spinlock than it would be waiting for the lock to clear.

# Properties of Spinlocks

It is important to note the differences in the various types of spinlocks.

Properties of kernel spinlocks:

- can have nested ownership.
- can use a level to enforce a lock hierarchy.
- can not be owned while outside of the kernel.
- can only be owned for very short periods of time.
- can not block while owning a spinlock.

Properties of device driver spinlocks:

- can't have nested ownership.
- can't use a level to enforce a lock hierarchy.
- can be held outside of the kernel.
- can only be owned for very short periods of time.

There is a different type of spinlock which is exported to subsystems. These locks are used to provide an efficient MP safe CLI/STI substitute for protecting data structures that are shared between task-time and interrupt-time code.

Properties of subsystem spinlocks:

- can't have nested ownership.
- can't use a level to enforce a lock hierarchy.
- can be held outside of the kernel.
- can only be used for very short periods of time.
- each processor can hold only one subsystem spinlock at a time.

A suspend lock, is defined such that an attempt to acquire a suspend lock which is currently owned by another processor places the current thread into a blocked state, and causes a reschedule. The thread's which are blocked on suspend locks will awaken when the lock is released. Suspend locks are only used inside the kernel.

Properties of a kernel suspend lock:

- can have nested ownership.
- can use a level to enforce a lock hierarchy.
- can not be owned while outside of the kernel.
- can be owned for long periods of time.
- can not block while owning a suspend lock.

When a spinlock is acquired, the lock manager automatically saves the state of the interrupt flag, then disables interrupts before returning to the caller. It restores the state of the interrupt flag when the lock is released. The kernel will panic if an interrupt is taken while owning a spinlock.

# Spinlock Use Guidelines

Here are some guidelines on using spinlocks to protect critical resources.

- Define spinlocks only for critical resources. A read only I/O port is not a critical resource. A set of read only I/O ports that must all be read before a decision is made IS a critical resource.

- Do not define too many spinlocks.

- Use spinlocks for VERY SHORT durations only. As a general rule, calls should be avoided while owning a spinlock.

- Leave interrupts disabled after locking a spinlock. This prevents interrupts on the same processor and possible deadlock.

- Be careful to not make any calls that may try to lock a spinlock that is already locked. ADDs, when making asynchronous callbacks, can be reentered at their IORB entry point.

- Be aware that DevHelp_Block called with spinlocks locked will unlock them. When the block wakes up spinlocks must be reacquired.

- Never make a call that could block while owning a spinlock. For example, some DevHelp calls can block.

# Device Drivers In OS/2 for SMP V2.11

This chapter describes the impacts to driver writers when writing device drivers for OS/2 for SMP V2.11.

Existing device drivers should run on OS/2 for SMP V2.11 without modifications providing two simple rules are followed:

- The driver must call DevHlp EOI to perform an EOI.
- The driver must not mask or unmask interrupts directly.

OS/2 2.x device drivers were written with only 8259 architecture in mind. The code that is most commonly executed in a device driver is to send the End Of Interrupt (EOI) command to the 8259. There is a system interface for do this, however, for performance reasons some device drivers have decided to implement this function directly. Additionally, some device drivers may MASK or UNMASK interrupts or read the 8259 registers to determine the interrupt state.

# Device Driver Spinlocks

The device driver should protect access to critical resources using spinlocks. The device driver allocates spinlocks in the Init routine by calling DevHlp_CreateSpinLock. CreateSpinLock returns a handle for use by the device driver. This handle is passed to DevHlp_AcquireSpinLock and DevHlp_ReleaseSpinLock. The spinlock is freed by calling DevHlp_FreeSpinLock. The driver may request any number of spinlocks, as the spinlocks are represented by a very small data structure. Once created, the spinlocks never go away.

As was outlined previously, OS/2 for SMP V2.11 contains a layer of abstraction for any functions that are deemed platform specific. These functions are placed inside the Platform Specific Driver (PSD) and isolate device drivers from the particular hardware platform that they are running on. At boot time, OS/2 determines and loads the appropriate PSD for the MP system hardware it is currently running on.

All device drivers that are MP-safe must use the appropriate kernel services to do hardware specific functions. The kernel will route these requests to the PSD for processing.

Device drivers in OS/2 2.x were written with the concept that only one processor can generate interrupts. But with OS/2 for SMP V2.11 other processors can now generate interrupts, so device drivers should account for re-entrance and parallel execution of task-time and interrupt-time code.

# Application Considerations

The following sections discuss application considerations of OS/2 for SMP V2.11.

# Application Compatibility Requirements

- An Application or associated subsystem must not use the 'INC' instruction as a semaphore without prepending a 'LOCK' prefix. On a UniProcessor (UP) system this instruction can be used as high performance semaphore without calling any other OS service if the semaphore is free and when the semaphore is clear and there are no waiters for the semaphore. Because the INC instruction can not be interrupted once started and because the results would be stored in the flags register which are per thread then it could be used safely as semaphore.

  In an OS/2 for SMP V2.11 environment this technique will not work because it is possible that two or more threads could be executing the same 'INC' instruction receiving the same results in each processor's/thread's flag register thinking that they each have the semaphore.

- Similarly a 486 or greater instruction such as the CMPXCHG has the same problem above if a 'LOCK' prefix is not prepended before the instruction.

- An Application or associated subsystem which relies on priorities to guarantee execution of its threads within a process will not work in OS/2 for SMP V2.11. For example an application may have a time-critical and an idle thread and may assume that while the time-critical thread is executing that the idle thread will not get any execution time unless the time-critical thread explicitly yields the CPU. In an OS/2 for SMP V2.11 environment it is possible that both the time-critical and idle threads are executing simultaneously on different processors.

The above compatibility requirements apply only to multithreaded applications, and therefore do not apply to DOS and WINOS2 applications. However, you are strongly encouraged to write 32-bit multithreaded applications for better performance and portability on OS/2 for SMP V2.11.

Given that there is the possibility of some set of applications which may use one of these techniques, OS/2 for SMP V2.11 provides a mechanism whereby these multithreaded applications can execute in UP mode. Only one thread of that process would be allowed to execute at any given time. That thread could execute on any one of the processors. A utility is used to mark the EXE file as uniprocessor only. OS/2 forces the process to run in the uniprocessor mode when the loader detects that the EXE file has been marked as uniprocessor only. See "The Single Processor Utility Program" section.

# Application Exploitation

There are some very attractive benefits of OS/2 for SMP V2.11 beyond the increased raw CPU power. Caching is a technique that is employed in both hardware and software to increase performance. SMPs increase the effectiveness of the various caches dramatically. An application that can divide its work into separate executing units such as threads will see performance increases across the hardware and software.

Each x86 processor (assuming 386 or higher) has a translation lookaside buffer (TLB) that keeps the most recent page translation addresses in a cache, so that every time the processor needs to translate a linear address into a physical address it does not have access the Page Directory and Page Table which reside in much slower memory. This cache is very limited in size. The more unique entries it encounters the less its effectiveness. An application which is single threaded makes use of only one TLB and probably causes thrashing within the TLB because of branching. However, with multiple processors, multithreaded applications will make use of N TLBs (where N is the number of threads and processors available). Thus the performance increase is more than just raw CPU power.

Beyond the TLB cache, these processors also contain Level 1 (L1) caches and OEMs will sometimes add Level 2 (L2) caches to their systems. The same advantages are applicable here but to a further degree.

There are also some advantages for software caches as well. Consider a file system cache where the effectiveness of the cache is largely determined by the hit ratio. If the cache receives large number of hits compared to misses, it is effective. The best way to achieve this is to keep the Most Recently Used (MRU) data in the cache. The best way to achieve this is to keep referencing the same data. A multithreaded application running on OS/2 for SMP V2.11 will cause this behavior to exist because the file system cache is being accessed in a shorter period of time by the same application. A single-threaded application with longer periods of access could allow for the cache to be flushed.

Secondly, an important aspect of a demand paged OS is its ability to keep the right set of pages in memory at the right times. With OS/2 for SMP V2.11 and a multithreaded application, the Page Manager can make a better decision because pages for this application are being accessed more frequently than before.

# New OS/2 for SMP V2.11 APIs

The new OS/2 for SMP V2.11 APIs are described in the following text.

This section defines the new spinlock APIs that have been added for multiprocessor support.

# DosCreateSpinLock

**Description**

Create a spinlock for multiprocessor serialization

**Calling Sequence**

APIRET DosCreateSpinLock (PHSPINLOCK pHandle)

**Parameters**

**pHandle** (*PHSPINLOCK*) - output

A pointer to the spinlock handle. This handle can be passed to DosAcquireSpinLock to acquire a spinlock and to DosReleaseSpinLock to release the spinlock.

**Returns**

**ulrc** (*APIRET*)

DosCreateSpinLock returns the following values:

| | |
|---|---|
| 0 | NO_ERROR |
| 32804 | ERROR_NO_MORE_HANDLES |

**Remarks**

DosCreateSpinLock returns a handle to a spin lock that is allocated in kernel data space. The handle is to be used on subsequent spin lock function calls and DevHlps.

**Related Functions**

- DosAcquireSpinLock
- DosReleaseSpinLock

**Example Code**

The following code example shows the use of DosCreateSpinLock:

```
#define INCL_BASE
#define OS2_API16
#define INCL_DOSSPINLOCK
#include <os2.h>
#include <stdio.h>
#include <string.h>
```

```
main()
{
    APIRET      rc;                    /* Return code */
    HSPINLOCK   Handle;                /* Handle to spin lock */
    PHSPINLOCK  pHandle = &Handle;     /* pointer to spin lock handle */

    /* Create a spin lock */

    rc = DosCreateSpinLock(pHandle);

    if (rc !=0)
      {
        printf("DosCreateSpinLock failed -- rc = %1d",rc);
        DosExit(0,1);
      }

    /* Acquire spin lock */

    rc = DosAcquireSpinLock(Handle);

    if (rc !=0)
      {
        printf("DosAcquireSpinLock failed -- rc = %1d",rc);
        DosExit(0,1);
      }

    /* Code that needs serialization */

    /* Release spin lock */

    rc = DosReleaseSpinLock(Handle);

    if (rc !=0)
      {
        printf("DosReleaseSpinLock failed -- rc = %1d",rc);
        DosExit(0,1);
      }
}
```

# DosAcquireSpinLock

**Description**

Acquire a spinlock for multiprocessor serialization

**Calling Sequence**

APIRET DosAcquireSpinLock (HSPINLOCK Handle)

**Parameters**

**Handle** (*HSPINLOCK*) - input

A handle to a spinlock. This handle was returned on the DosCreateSpinLock api call.

**Returns**

**ulrc** (*APIRET*)

DosAcquireSpinLock returns one of the following values:

| | |
|---|---|
| 0 | NO_ERROR |
| 6 | ERROR_INVALID_HANDLE |

**Remarks**

DosAcquireSpinLock is passed a handle which was returned by DosCreateSpinLock When control is returned to the requester, the spin lock has been acquired and interrupts are disabled. A call to DosReleaseSpinLock must follow very shortly. Spin locks can be nested.

**Related Functions**

- DosCreateSpinLock
- DosReleaseSpinLock

**Example Code**

The following code example shows the use of DosAcquireSpinLock

```
#define INCL_BASE
#define OS2_API16
#define INCL_DOSSPINLOCK
#include <os2.h>
#include <stdio.h>
#include <string.h>
```

```
main()
{
    APIRET      rc;                       /* Return code */
    HSPINLOCK   Handle;                   /* Handle to spin lock */
    PHSPINLOCK  pHandle = &Handle;  /* pointer to spin lock handle */

    /* Create a spin lock */

    rc = DosCreateSpinLock(pHandle);

    if (rc !=0)
      {
        printf("DosCreateSpinLock failed -- rc = %1d",rc);
        DosExit(0,1);
      }

    /* Acquire spin lock */

    rc = DosAcquireSpinLock(Handle);

    if (rc !=0)
      {
        printf("DosAcquireSpinLock failed -- rc = %1d",rc);
        DosExit(0,1);
      }

    /* Code that needs serialization */

    /* Release spin lock */

    rc = DosReleaseSpinLock(Handle);

    if (rc !=0)
      {
        printf("DosReleaseSpinLock failed -- rc = %1d",rc);
        DosExit(0,1);
      }
}
```

# DosReleaseSpinLock

**Description**

Release a spinlock for multiprocessor serialization

**Calling Sequence**

APIRET DosReleaseSpinLock (HSPINLOCK Handle)

**Parameters**

**Handle** (*HSPINLOCK*) - input

A handle to a spinlock. This handle was returned by DosCreateSpinLock.

**Returns**

**ulrc** (*APIRET*)

DosReleaseSpinLock returns the following values:

| | |
|---|---|
| 0 | NO_ERROR |
| 6 | ERROR_INVALID_HANDLE |

**Remarks**

DosReleaseSpinLock is passed a handle which was returned by DosCreateSpinLock When control is returned to the requester, the spin lock is released and interrupts are enabled. A DosAcquireSpinLock must have been previously issued.

**Related Functions**

- DosAcquireSpinLock
- DosCreateSpinLock

**Example Code**

The following code example shows the use of DosReleaseSpinLock:

```
#define INCL_BASE
#define OS2_API16
#define INCL_DOSSPINLOCK
#include <os2.h>
#include <stdio.h>
#include <string.h>
```

```c
main()
{
    APIRET      rc;                          /* Return code */
    HSPINLOCK   Handle;                      /* Handle to spin lock */
    PHSPINLOCK  pHandle = &Handle;  /* pointer to spin lock handle */

    /* Create a spin lock */

    rc = DosCreateSpinLock(pHandle);

    if (rc !=0)
      {
        printf("DosCreateSpinLock failed -- rc = %1d",rc);
        DosExit(0,1);
      }

    /* Acquire spin lock */

    rc = DosAcquireSpinLock(Handle);

    if (rc !=0)
      {
        printf("DosAcquireSpinLock failed -- rc = %1d",rc);
        DosExit(0,1);
      }

    /* Code that needs serialization */

    /* Release spin lock */

    rc = DosReleaseSpinLock(Handle);

    if (rc !=0)
      {
        printf("DosReleaseSpinLock failed -- rc = %1d",rc);
        DosExit(0,1);
      }
}
```

# DosFreeSpinLock

**Description**
Free a spinlock for multiprocessor serialization.

**Calling Sequence**
APIRET DosFreeSpinLock (HSPINLOCK Handle)

**Parameters**
**Handle** (*HSPINLOCK*) - input
A handle to a spinlock. This handle was returned on the DosCreateSpinLock api call.

**Returns**
**ulrc** (*APIRET*)
DosFreeSpinLock returns the following values:

0 NO_ERROR
6 ERROR_INVALID_HANDLE

**Remarks**
DosFreeSpinLock is passed the handle which was returned by DosCreateSpinLock

**Related Functions**

- DosCreateSpinLock
- DosAcquireSpinLock
- DosReleaseSpinLock

**Example Code**
The following code example shows the use of DosFreeSpinLock:

```
#define INCL_BASE
#define OS2_API16
#define INCL_DOSSPINLOCK
#include <os2.h>
#include <stdio.h>
#include <string.h>

main()
```

```
{
    APIRET      rc;                         /* Return code */
    HSPINLOCK   Handle;                     /* Handle to spin lock */
    PHSPINLOCK  pHandle = &Handle;  /* pointer to spin lock handle */

    /* Create a spin lock */

    rc = DosCreateSpinLock(pHandle);
    if (rc !=0)
      {
       printf("DosCreateSpinLock failed -- rc = %1d",rc);
       DosExit(0,1);
      }

    /* Acquire spin lock */

    rc = DosAcquireSpinLock(Handle);
    if (rc !=0)
      {
       printf("DosAcquireSpinLock failed -- rc = %1d",rc);
       DosExit(0,1);
      }

    /* Code that needs serialization */
    /* Release spin lock */

    rc = DosReleaseSpinLock(Handle);
    if (rc !=0)
      {
       printf("DosReleaseSpinLock failed -- rc = %1d",rc);
       DosExit(0,1);
      }

    /* Free spin lock */

    rc = DosFreeSpinLock(Handle);
    if (rc !=0)
      {
       printf("DosFreeSpinLock failed -- rc = %1d",rc);
       DosExit(0,1);
      }
}
```

New APIs are being introduced to provide support for the OS/2 SMP V2.11 performance monitor.

# DosGetProcessorCount

**Description**

Get the count of usable processors

**Calling Sequence**

```
APIRET DosGetProcessorCount (PULONG pCount)
```

**Parameters**

**pCount** (*PULONG*) - output
A pointer to the count of usable processors.

**Returns**

DosGetProcessorCount returns the following values:

| | |
|---|---|
| 0 | NO_ERROR |
| 87 | ERROR_INVALID_PARAMETER |

**Remarks**

DosGetProcessorCount returns the number of usable processors.

**Related Functions**

- DosGetProcessorIdleTime
- DosGetProcessorStatus
- DosSetProcessorStatus

# DosGetProcessorIdleTime

**Description**

Get the idle time for the specified processor.

**Calling Sequence**

```
APIRET DosGetProcessorIdleTime (ULONG ProcNum, PULONG pIdleTime)
```

**Parameters**

**ProcNum** (*ULONG*) - input
The processor number for which the idle time is to be gotten.

**pIdleTime** (*PULONG*) - output
A pointer to the idle time for the specified processor.

**Returns**

DosGetProcessorIdleTime returns the following values:

| | |
|---|---|
| 0 | NO_ERROR |
| 87 | ERROR_INVALID_PARAMETER |

**Remarks**

DosGetProcessorIdleTime returns the idle time for the specified processor

**Related Functions**

- DosGetProcessorCount
- DosGetProcessorStatus
- DosSetProcessorStatus

# DosGetProcessorStatus

**Description**

Get the status for the specified processor.

**Calling Sequence**

```
APIRET DosGetProcessorStatus (ULONG ProcNum, PULONG pStatus)
```

**Parameters**

**ProcNum** (*ULONG*) - input
The processor number for which the status is to be gotten.

**pStatus** (*PULONG*) - output
A pointer to the status for the specified processor.

**Returns**

DosGetProcessorStatus returns the following values:

| | |
|---|---|
| 0 | NO_ERROR |
| 87 | ERROR_INVALID_PARAMETER |

**Remarks**

DosGetProcessorStatus returns the status for the specified processor. A 0 indicates OFFLINE and a 1 indicates ONLINE. All other values are reserved.

**Related Functions**

- DosGetProcessorCount
- DosGetProcessorIdleTime
- DosSetProcessorStatus

# DosSetProcessorStatus

**Description**

Set the status for the specified processor.

**Calling Sequence**

```
APIRET DosSetProcessorStatus (ULONG ProcNum, PULONG pStatus)
```

**Parameters**

**ProcNum** (*ULONG*) - input
The processor number for which the status is to be set.

**pStatus** (*PULONG*) - input
A pointer to the status for the specified processor.

**Returns**

DosSetProcessorStatus returns the following values:

| | |
|---|---|
| 0 | NO_ERROR |
| 87 | ERROR_INVALID_PARAMETER |

**Remarks**

DosSetProcessorStatus sets the status for the specified processor. A 0 indicates OFFLINE and a 1 indicates ONLINE. All other values are reserved.

**Related Functions**

- DosGetProcessorCount
- DosGetProcessorIdleTime
- DosGetProcessorStatus

# DosAllocThreadLocalMemory

**Description**

Allocates a block of memory that is local to a thread.

**Calling Sequence**

```
APIRET DosAllocThreadLocalMemory (ULONG Lwords, PPVOID pMemBlock)
```

**Parameters**

**Lwords** (*ULONG*) - input
The number of 32-bit dwords to allocate.

**pMemBlock** (*PPVOID*) - input
A pointer to the memory block allocated.

**Returns**

DosAllocThreadLocalMemory returns the following values:

| | |
|---|---|
| 0 | NO_ERROR |
| 8 | ERROR_NOT_ENOUGH_MEMORY |
| 87 | ERROR_INVALID_PATAMETER |

**Remarks**

When a process is started, a small block of memory is set aside to be used as a thread-local memory area. This memory is at the same virtual address for each thread, but is backed by different physical memory. This permits each thread to have a small block of memory that is unique, or local, to that thread.

The thread-local memory area consists of 32 DWORDs (128 bytes), each DWORD being 32-bits in size. Up to 8 DWORDs (32 bytes) can be requested each time this function is called. If you want to allocate more than 8 DWORDs, you must call this function more than once.

Allocation is by DWORD only. If you want to store a BYTE in the thread-local memory area, you would still allocate a DWORD, then store the BYTE in it.

**Related Functions**

- DosFreeThreadLocalMemory

**Example Code**

The following code example allocates a thread-local memory block of 6 DWORDs, then frees it.

```
#define  INCL_DOSPROCESS  /* Memory Manager values */
#include <os2.h>

#include <stdio.h>  /* For printf */

PVOID    pMemBlock;     /* Pointer to the memory block returned */
APIRET   rc;            /* Return code */

   rc = DosAllocThreadLocalMemory(6, &pMemBlock);   /* Allocate 6 DW

   if (rc != NO_ERROR)
   {
       printf("DosAllocThreadLocalMemory error: return code = %ld", 
       return 1;
   }


   /* ... Use the thread-local memory block ... */


   rc = DosFreeThreadLocalMemory(pMemBlock);     /* Free the memory 

   if (rc != NO_ERROR)
   {
       printf("DosFreeThreadLocalMemory error: return code = %ld", r
       return 1;
   }

   return 0;
```

# DosFreeThreadLocalMemory

**Description**

Free memory allocated by DosAllocThreadLocalMemory.

**Calling Sequence**

APIRET DosFreeThreadLocalMemory (ULONG ProcNum, PULONG pStatus)

**Parameters**

**ProcNum** (*ULONG*) - input
The processor number for which the status is to be set.

**pStatus** (*PULONG*) - input
A pointer to the status for the specified processor.

**Returns**

DosFreeThreadLocalMemory returns the following values:

| | |
|---|---|
| 0 | NO_ERROR |
| 87 | ERROR_INVALID_PARAMETER |

**Remarks**

When a process is started, a small block of memory is set aside to be used as a thread-local memory area. This memory is at the same virtual address for each thread, but is backed by different physical memory. This permits each thread to have a small block of memory that is unique, or local, to that thread.

The thread-local memory area consists of 32 DWORDs (128 bytes), each DWORD being 32-bits in size.

**Related Functions**

- DosAllocThreadLocalMemory

**Example Code**

The following code example allocates a thread-local memory block of 6 DWORDs, then frees it.

```
#define  INCL_DOSPROCESS  /* Memory Manager values */
#include <os2.h>
```

```
#include <stdio.h>  /* For printf */

PVOID    pMemBlock;  /* Pointer to the memory block returned */
APIRET   rc;         /* Return code */

   rc = DosAllocThreadLocalMemory(6, &pMemBlock); /* Allocate 6 DWORD

   if (rc != NO_ERROR)
   {
       printf("DosAllocThreadLocalMemory error: return code = %ld", r
       return 1;
   }


   /* ... Use the thread-local memory block ... */

   rc = DosFreeThreadLocalMemory(pMemBlock); /* Free the memory block

   if (rc != NO_ERROR)
   {
       printf("DosFreeThreadLocalMemory error: return code = %ld", rc
       return 1;
   }

   return 0;
```

# DosQuerySysInfo

DosQuerySysInfo now returns three new system variables. These variables are in the following list.

| VALUE | MNEMONIC CONSTANT AND DESCRIPTION |
|---|---|
| 24 | QSV_FOREGROUND_FS_SESSION<br>Session ID of the current foreground full-screen session. Note that this only applies to full-screen sessions. The Presentation Manager session (which displays VIO-windowed, PM, and windowed DOS Sessions) is full-screen session ID 1. |
| 25 | QSV_FOREGROUND_PROCESS<br>Process ID of the current foreground process. |
| 26 | QSV_NUMPROCESSORS<br>Number of processors in the machine. |

# Avoiding Device Driver Deadlocks

Deadlock can be defined as an unresolved contention for use of a resource. Whenever any mutual exclusion primitive is used, the possibility of deadlock is introduced. This is evident even in uniprocessor system such as OS/2 with the use of semaphores. The possibilities of deadlock are greater in a multiprocessor environment because of the large requirement for mutual exclusion. The method of mutual exclusion for device drivers and the OS/2 SMP kernel is the spinlock. Using spinlocks incorrectly can result in deadlock conditions where an application or device driver will become hung. In the case of a device driver, no more activity will take place on that processor if the device driver enters a deadlock state. Writing device drivers and code for OS/2 for SMP V2.11 requires the programmer to think about the conditions in the code which might cause a deadlock condition, and then use spinlocks to protect those resources.

While it would be impossible to list every cause of deadlock, a few of the most common code examples are given below in pseudo-code that can result in deadlock. These examples are not exhaustive, but represent the majority of situations that will probably be encountered. Being aware of these types of conditions can help you reduce the chances of deadlock within your device driver or applications.

## Use of CLI/STI

As stated above, CLI/STI will only work on the processor on which they execute. Therefore, only the same processor will be protected from "stepping" on a protected resource. For example, assume the application maintains a linked list of I/O packets for a device. Whenever packets are inserted or removed, the list must be protected as a critical resource. Under the uniprocessor model, a CLI/STI around the manipulation of the list would be sufficient protection. However, in an MP environment, the CLI/STI would only protect the resource on the same processor. Another processor could enter a section of code that attempted to manipulate the linked list. The results would be unpredictable. Possibilities would range from no effect to deadlock. Code that uses CLI/STI is not reliable and should be eliminated.

The solution is to replace CLI/STIs with spinlocks. Each critical resource will have associated with it a spinlock. Before accessing the resource the spinlock must be acquired, and when complete, the spinlock is released.

## Spinlocks Taken Out of Order

One possible cause of deadlock stems from taking spinlocks in different orders in different sections of code. Consider the following two sections of code, each executing on a separate processor at the same time. For both examples all locks are available when the code begins execution.

```
   Code section 1

1  Lock spinlock1
2  Do some processing
3  Lock spinlock2
4  More processing
5  Unlock spinlock2
6  Unlock spinlock1
```

```
   Code section 2

1  Lock spinlock2
2  Do some processing
3  Lock spinlock1
4  More processing
5  Unlock spinlock1
6  Unlock spinlock2
```

In section 1 line 1 locks spinlock1. In section 2 line 1 locks spinlock2. Both sections will successfully lock their respective locks and continue normally. Now section 1 on line 3 tries to lock spinlock2, which is already locked by section 2, so section 1 spins. Now section 2 tries to lock spinlock1 (line 3), which is already locked by section 1, so section 2 now spins. Now each section of code is spinning waiting for a lock that the other owns. The result is deadlock. Neither section of code will ever continue executing and will therefore never release the spinlock that the other needs. This kind of deadlock is very common, but can be avoided by always taking spinlocks that are related in the same order.

To fix the above code, code section 2 would be recoded to the following:

```
   Code section 2

1  Lock spinlock1
2  Lock spinlock2
3  Do some processing
4  More processing
5  Unlock spinlock2
6  Unlock spinlock1
```

By taking the locks in the same order as code section 1 the deadlock potential is

eliminated. Both sections can no longer be waiting on a resource the other owns at the same time. It should be noted that spinlocks should be released in the reverse order that they are locked.

# Blocking With Spinlocks Locked

Another cause of deadlock is blocking with locked spinlocks. Consider the following two sections of code. Section 1 is a task time operation that needs an interrupt to complete. Section 2 is the interrupt code that will execute and unblock section 1.

```
Code section 1                  Code section 2
(Task time)                     (Interrupt time)

Lock spinlock1                  interrupt received
start I/O                       lock spinlock1
block (ProcBlock)               unblock (ProcRun)
                                release spinlock1
return from block
some processing
   (may include a re-block)
release spinlock1
```

In the above example code section 1 locks spinlock1 and then blocks (with the spinlock still locked). Code section 2 will then execute when the I/O completes. The interrupt code first tries to lock spinlock1. Because spinlock1 is already locked, the interrupt code will spin waiting for the lock. The lock will never become available, however, because the only way for the spinlock to be unlocked is for section 1 to be unblocked. But the interrupt code, which is responsible for the unblock, can't continue until it acquires the spinlock. The result is deadlock.

Now the first attempt to solve this problem may be to recode section 1 with the following:

```
Lock spinlock1
start I/O
release spinlock1
block (ProcBlock)

return from block
lock spinlock1
some processing
release spinlock1
```

The above code sequence appears to correct the problem. It does not, however, and can also result in a deadlock. The reason is that there exists a window between where the code releases the spinlock and the thread is blocked in which an interrupt can occur. Remember that disabling interrupts no longer prevents interrupts from happening. If an interrupt fires in this window, the interrupt handler (section 2 above) will run. It will acquire the spinlock and attempt to unblock the thread. The thread, however, has not actually blocked yet. When the thread finally does block, the wakeup event has already occurred. The result once again is deadlock.

To solve this particular problem, DevHelp_Block has been modified to release ALL spinlocks that are owned on the current processor. The device driver should call DevHelp_Block with spinlocks locked. The kernel will first put the thread of execution in the blocked state. Then, before dispatching the next thread, it will release all locked spinlocks for the current processor. Because the thread is in the blocked state, it is valid for another processor to execute interrupt code that will do the DevHelp_Run. The result is no deadlock. The code sequences from above should be re-coded to the following to avoid the deadlock:

| Code section 1 | Code section 2 |
| --- | --- |
| Lock spinlock1 | interrupt received |
| start I/O | lock spinlock1 |
| While(block required) | unblock (ProcRun) |
|    Block | release spinlock1 |
|    return from block | |
|    Lock spinlock1 | |
| EndWhile | |
| some processing | |
| release spinlock1 | |

The above example has been expanded to include the steps required to insure that when the thread is woken up, that the blocking condition is satisfied before execution continues. This code sequence is analogous to that listed in the description for DevHlp_Block in the Device Helper Services chapter of the Physical Device Driver Reference. It has been modified to use spinlocks instead of disabling interrupts (which will not work).

Once again this list is not exhaustive, but is a representation of the majority of cases that can cause deadlock. By avoiding these situations the chances of deadlock are reduced considerably. In addition, there are certain system level checks performed to help insure that deadlock is avoided. If the system detects a situation that could cause deadlock, such as attempting to block while owing a spinlock, it will panic the system and print an internal processing error message.

## Blocking

As shown in the last example, there are special considerations that must be followed when blocking in an MP aware device driver. Because blocking with a spinlock owned can cause deadlock, the DevHelp_Block service will unlock spinlocks as part of the blocking sequence. When the run is done and the blocked thread begins execution again, it must again lock any required spinlocks.

All system components that use spinlocks must be aware of calls that may block. For example, the file system, which calls a device driver to perform I/O, will almost always block in the device driver. The file system therefore should release all spinlocks before calling the device driver. In general, release all spinlocks before making a call that could block.

# Interrupt Processing

Interrupt processing should not be affected, except by the need to lock spinlocks for critical resources. When a spinlock is locked, the LockManger will disable interrupts before returning to the device driver. This insures that no interrupt will occur, on the same processor, between when the spinlock is requested and when the kernel returns to the device driver with the spinlock locked. (The same level of function accomplished by a CLI on a single processor system). The device driver MUST leave interrupts disabled while owning the spinlock. If interrupts were enabled a deadlock could occur.  Consider the following:

```
   Task Time                             Int Time

(ints enabled)
Lock spinlock1
STI
                ---Interrupt--->     Lock spinlock1
                                     some processing
                                     Unlock spinlock1
                                     EOI
                <-- Return from Int
Some processing
Unlock spinlock1
```

In the above example the the task time and interrupt code are running on the same processor. When the task time code locks spinlock1 with interrupts enabled the LockManager will return with interrupts disabled. If interrupts were enabled after the lock with the STI instruction, then the interrupt code on the right could run. The first thing the interrupt handler does is try to grab spinlock1. Because spinlock1 is already locked, the interrupt handler will spin. The lock, however, will never become available. The task time code will not run until the interrupt code completes. The result is deadlock. This is why it is important to leave interrupts disabled while owning a spinlock.

Consider the same code above, but with the task time code running on processor A and the interrupt code running on processor B. For this example, however, interrupts remain disabled (remove the STI). Because the LockManager disables interrupts, processor B will run the interrupt code. When the interrupt code attempts to get the spinlock, it will spin. Because processor A continues executing, the spinlock will be released, thereby allowing the interrupt code on processor B to acquire the spinlock and continue execution.  Deadlock is avoided. When processor A returns from the unlock the state of the interrupt flag will be restored by the LockManager to its state before the lock was done.

Another action the device driver must avoid is issuing its own EOI. All EOIs must use the DevHelp_EOI device helper service. The reason for this is that different multiprocessor platforms have defined their own advanced interrupt controllers. Without detailed knowledge of the controller and how it operates, and knowledge of how the kernel is using the controller, the device driver can cause unpredictable results, including deadlock. All MP-aware device drivers must use the EOI service.

# New Device Helper (DevHlp) Routines

Following are the new physical and virtual DevHlp routines.

# Physical DevHlps

The OS/2 kernel will provide new DevHlps for OS/2 for SMP V2.11 as follows.

```
DevHlp_CreateSpinLock         EQU    111     ; 6F Create a spinlock - SMP
DevHlp_FreeSpinLock           EQU    112     ; 70 Free a spinlock - SMP
DevHlp_AcquireSpinLock        EQU    113     ; 71 Acquire a spinlock - SMP
DevHlp_ReleaseSpinLock        EQU    114     ; 72 Release a spinlock - SMP
DevHlp_PortIO                 EQU    118     ; 76 Port I/O
DevHlp_SetIRQMask             EQU    119     ; 77 Set/Unset an IRQ mask
DevHlp_GetIRQMask             EQU    120     ; 78 Get an IRQ mask
```

# DevHlp_CreateSpinLock

**Description**
Create a subsystem spinlock.

This function creates a subsystem spinlock.

**Parameters**
Pointer to spinlock handle.

**Exit**
Return code.

**Assembly language**

```
;       dh_CreateSpinLock - Create a spinlock
;
;       This routine creats a subsystem spinlock.
;
;       ENTRY:  AX:BX = pointer to store spinlock handle
;
;       EXIT:   None
;
;       USES:   EAX, Flags
;

  MOV     AX,AddressHigh              ; high word of address
  MOV     BX,AddressLow               ; low word of address
  MOV     DL,DevHlp_CreateSpinLock    ;
  CALL    DevHlp
  JC      Error
```

# DevHlp_FreeSpinLock

**Description**

Free a subsystem spinlock.

This function frees a subsystem spinlock.

**Parameters**

Spinlock handle.

**Exit**

Return code.

**Assembly language**

```
;       dh_FreeSpinLock - Free a subsystem spinlock
;
;       This routine frees a subsystem spinlock.
;
;       ENTRY:  AX:BX = spinlock handle
;
;       EXIT:   None
;
;       USES:   Flags
;

  hSpinLock       dd      ?               ; 16:16

  MOV     AX,hSpinLockHighWord            ; high word of handle
  MOV     BX,hSpinLockLowWord             ; low word of handle
  MOV     DL,DevHlp_FreeSpinLock          ;
  CALL    DevHlp
  JC      Error
```

# DevHlp_AcquireSpinLock

**Description**

Acquire a subsystem spinlock.

This function obtains ownership of a subsystem spinlock.

**Parameters**

Spinlock handle.

**Exit**

Return code.

**Assembly language**

```
;       dh_AcquireSpinLock - Acquire a subsystem spinlock
;
;       Obtains ownership of a subsystem spinlock. Used by device dri
;
;       ENTRY:  AX:BX = spinlock handle
;
;       EXIT:   None
;
;       USES:   Flags
;

  hSpinLock      dd      ?            ; 16:16

  MOV    AX,hSpinLockHighWord         ; high word of handle
  MOV    BX,hSpinLockLowWord          ; low word of handle
  MOV    DL,DevHlp_AcquireSpinLock    ;
  CALL   DevHlp
  JC     Error
```

# DevHlp_ReleaseSpinLock

**Description**

Release a subsystem spinlock.

This function releases ownership of a subsystem spinlock.

**Parameters**

Spinlock handle.

**Exit**

Return code.

**Assembly language**

```
;       dh_ReleaseSpinLock - Release a subsystem spinlock.
;
;       Releases ownership of a subsystem spinlock. Used by device dr
;
;       ENTRY:  AX:BX = spinlock handle
;
;       EXIT:   None
;
;       USES:   Flags
;

  hSpinLock       dd      ?             ; 16:16

  MOV     AX,hSpinLockHighWord          ; high word of handle
  MOV     BX,hSpinLockLowWord           ; low word of handle
  MOV     DL,DevHlp_ReleaseSpinLock     ;
  CALL    DevHlp
  JC      Error
```

# DevHlp_Port_IO

**Description**

Perform IO to a specified port.

This function is used to perform input/output operations to a specified local port.

**Parameters**

Pointer to a PORT_IO structure.

**Exit**

Return code.

**Structures**

```
typedef struct port_io_s
{
   ulong_t port;    (Input)
   ulong_t data;    (Input/Output)
   ulong_t flags;   (Input)
} PORT_IO;
```

| | |
|---|---|
| port | indicates which port to read to, or write from. |
| data | contains the data read from a read request, or the data to write if a write request. |
| flags | indicate what operation to perform. |

| | |
|---|---|
| IO_READ_BYTE | Read a byte from the port |
| IO_READ_WORD | Read a word from the port |
| IO_READ_DWORD | Read a dword from the port |
| IO_WRITE_BYTE | Write a byte to the port |
| IO_WRITE_WORD | Write a word to the port |
| IO_WRITE_DWORD | Write a dword to the port |

**Assembly language**

```
;       dh_Port_IO - Perform I/O to a specified port
;
;       This devhlp is called by device drivers to do
;       I/O to a specified local port.
;
;       ENTRY:  ES:DI = pointer to port_io structure
```

```
;
;       EXIT:   port_io.data filled in if I/O read
;
;       USES:   EAX, Flags
;

port_io_s          STRUC
port_io_port         DD   ?
port_io_data         DD   ?
port_io_flags        DD   ?
port_io_s          ENDS

IO_READ_BYTE       EQU  0000H
IO_READ_WORD       EQU  0001H
IO_READ_DWORD      EQU  0002H
IO_WRITE_BYTE      EQU  0003H
IO_WRITE_WORD      EQU  0004H
IO_WRITE_DWORD     EQU  0005H
IO_FLAGMASK        EQU  0007H

MOV     PORT_IO.port_io_port,21h
MOV     PORT_IO.port_io_data,08h
MOV     PORT_IO.port_io_flags,IO_WRITE_BYTE

LES     SI,PORT_IO
MOV     DL,dh_Port_IO
CALL    DevHlp
JC      Error

;       EXIT:   port_io_struc.data filled in if I/O read
```

**Notes**

None.

# DevHlp_SetIRQMask

**Description**

Enable/disable interrupt.

This function enables and/or disables interrupts for a specific IRQ.

**Parameters**

Specified IRQ level.
Enable/disable flag.

**Exit**

Return code.

**Assembly language**

```
;       dh_SetIRQMask - Masks/Unmasks a specified IRQ masks
;
;       This function enables/disables interrupts for a specific IRQ.
;
;       ENTRY   AL = IRQ to be enabled/disabled
;               AH =  0  enable IRQ (disable mask)
;                     1  disable IRQ (enable mask)
;
;       EXIT-SUCCESS
;           none
;
;       EXIT-FAILURE
;           NONE
;

  MOV     AL,IRQ to enable/disabled
  MOV     AH,mask operation (0=enabled,1=disabled)
  MOV     DL,DevHlp_SetIRQMask
  CALL    DevHlp
  JC      Error
```

# DevHlp_GetIRQMask

**Description**

Retrieve the mask state of an IRQ level.

This function reads the current IRQ mask state for the specified IRQ.

**Parameters**

Specified IRQ level.

**Exit**

EAX=0, mask disabled (IRQ enabled).
EAX=1, mask enabled (IRQ disabled)

**Assembly language**

```
;       dh_GetIRQMask - Retrieve a specified IRQ mask state
;
;       This function reads the current IRQ mask state for the specif
;
;       ENTRY   AL = IRQ
;
;       EXIT-SUCCESS
;           EAX - 0 = mask disabled (IRQ enabled)
;               - 1 = mask enabled (IRQ disabled)
;
;       EXIT-FAILURE
;           NONE
;

  MOV     AL,IRQ whose mask state is to be retrieved
  MOV     DL,DevHlp_GetIRQMask
  CALL    DevHlp
  JC      Error
```

# Virtual Device Driver Helps

The OS/2 kernel will provide new VDH services for VDDs to communicate with PSDs.

# VDHPortIO

**Description**

Perform IO to a specified port.

This function is used to perform input/output operations to a specified local port.

**Parameters**

Pointer to a PORT_IO structure.

**Exit**

Return code.

**Structures**

```
typedef struct port_io_s
{
   ulong_t port;    (Input)
   ulong_t data;    (Input/Output)
   ulong_t flags;   (Input)
} PORT_IO;
```

| | |
|---|---|
| port | indicates which port to read to, or write from. |
| data | contains the data read from a read request, or the data to write if a write request. |
| flags | indicate what operation to perform. |

| | |
|---|---|
| IO_READ_BYTE | Read a byte from the port |
| IO_READ_WORD | Read a word from the port |
| IO_READ_DWORD | Read a dword from the port |
| IO_WRITE_BYTE | Write a byte to the port |
| IO_WRITE_WORD | Write a word to the port |
| IO_WRITE_DWORD | Write a dword to the port |

**Notes**

None.

# New Kernel Debugger Commands

The Kernel debugger architecture is such that only one thread can be in the debugger at any given time, so it uses a spinlock to serialize its access.

If entered, the debugger must inform the user as to the state of all the processors, even though the other processors are still executing code. It accomplishes this by sending a spin command using and IPI (interprocessor interrupt) to all the other processors. When a processor receives a spin command sent by the kernel debugger, it saves its current state (all of its registers), acknowledges the spin command, and spins until released. This allows the user to switch to a slot which is currently executing on another processor and determines what it is doing.

All kernel debugger commands work as before, but a few have been modified to display or use MP specific information, and new MP specific commands have been added.

A list of new and changed commands follows:

- A **.DP** (processor status) command has been added. This command dumps out a processor control block verbosely. As an argument it takes a * (real current slot), a # (currently selected slot), and a 1 based processor number (e.g. **.DP 3** displays the processor status for processor 3), or a blank (e.g. **.DP** ) which displays the processor status for all the processors.

- A **.DL** (display processor spinlocks) command has been added. This command displays all the spinlocks owned by a particular processor. As an argument it takes a * (real current slot), a # (currently selected slot), a 1 based processor number (e.g. **.DL 3** displays all the spinlocks owned by processor 3), an address of a spinlock, or a blank which displays all the spinlocks owned by all the processors.

- The **.R** and the **R** (register commands) have been modified to indicate which processor the currently selected slot is running on. A p=xxyy (xx = processor number, yy = flags) has been added to the end of the third register line. These processor numbers are 1- based (e.g. p=00 means that the currently selected slot is not running on any processor or is blocked, p=01 means the currently selected slot is running on processor 1). The flags are:

  **s** processor is currently spinning.
  **r** processor is attempting to grab the ring 0 suspend lock.

- The **.SS** (change current slot) has been modified to change which PSA (processor save area) you are currently looking at (e.g. when you change to a slot which is currently running on a different processor and dump a variable in the PSA, it will display the value of that variable on that particular processor). The **.S** command is now identical to the **.SS** command. The PLMA is

displayed properly for each processor.

# The Single Processor Utility Program

As explained previously, some applications written for uniprocessor OS/2 may experience problems running under OS/2 for SMP V2.11 because they rely upon priorities between threads for accessing shared resources, or use the CLI/STI method for protecting resources like semaphores or memory. These types of application are called MP-safe. These programs will still run fine under OS/2 for SMP V2.11 if they are run in a uniprocessor mode.

The EXECMODE program is a utility which marks the executable (EXE) file to be run in a uniprocessor mode. The OS/2 loader detects this bit set and forces the application to run on a single processor. The EXECMODE utility can be used to set and reset the uniprocessor mode in an executable file, as well as list those programs that are marked as MP or SP.

The syntax for the EXECMODE utility program is as follows:

```
execmode (options)[d:[\[path\]]]filenam1.ext( options) [filenam2.ext]...
```

The EXECMODE program accepts several command line options. Each option must be preceeded by a "/" or a "-".

| | |
|---|---|
| **sp** | Set file in single processor mode (default) |
| **mp** | Set file for multiprocessor mode |
| **l** | List files matching **sp** or **mp** |
| **s** | Enable subdirectory searching |
| **f** | Force changes on read-only files |
| **v** | Set verbose mode on |
| **q** | Set for quiet mode |
| **d** | Display debug messages |
| **t** | Set test mode (no disk writes) |

Up to 50 arguments, in any order, can be specified on the command line. Wildcards are permitted in filenames.

# OS/2 for SMP V2.11 Tools

A Multiprocessor CPU Performance Monitor will be shipped with this product. This tool will display CPU utilization for each processor in bar graph and histogram modes. It will be written as a PM application and will display each processor's bar or line as a different color. This tool will also have the capability of placing each processor offline or online. This is useful to show the scalability of OS/2 for SMP V2.11. It may also be used for debug purposes. This tool will use the APIs described above. It is also desirable to be able to display the time spent waiting inside of the major spinlocks, such as the Ring 0 spinlock. It is also desirable to display the interrupt activity for each processor.

```
     OS/2 Symmetric MultiProcessor Performance Monitor

    Bar  Histogram   Interrupt   Status  Options  Help



  100

   90

   80

   70

   60

   50

   40

   30

   20

   10
```

```
  0
 %
 CPU   1       2       3       4       5       6
```

Figure 1.       CPU Monitor in BAR mode

Monitors % of each processor used per second.

NOTE: Each CPU to be a different color.

OS/2 Symmetric MultiProcessor Performance Monitor

Bar   Histogram   Interrupt   Status   Options   Help

```
100

 90

 80

 70

 60

 50                                  ******
                   *  ***           *      *
               **** **   ***       *        *
 40   ***     *             *   * *          ****
    **   *   *               * * *               *
          ***                 *                   **
 30 x       -                                     x
     x  ---- -xxx                          xxx xxx x
      x-  xxxx-  xx  -    xxxx  -         x   x  ---
 20   -x x     ----x- ---x    x- - ------x      -
    --  x          -x   x---  -x  -  xxxx---   -
                     xxx    --  xxxxx       - -
 10                                          -
```

0
%/
/Time 1 2 3 4 5 6

Figure 2. CPU Monitor in HISTOGRAM mode

Monitors % of each processor used over time.

NOTE: Each CPU to be a different color.

OS/2 Symmetric MultiProcessor Performance Monitor

Bar Histogram Interrupt Status Options Help

5000

4500

4000

3500

3000

2500

2000

1500

1000

500

0
#/
/Time 1 2 3 4 5 6

Figure 3. CPU Monitor in INTERRUPT mode

Monitors # of interrupts per second.

NOTE: Each CPU to be a different color.

Processor ONLINE/OFFLINE STATUS Selection

Please select/change the status of the desired processor(s).

A Y means the processor is online
A N means the processor is offline

Selection toggles the Y/N

| CPU | ONLINE |
| --- | --- |
| | n X |
| 1 | Y |
| 2 | Y |
| 3 | N |
| 4 | Y |
| 5 | Y |
| 6 | N |
| 7 | N |
| 8 | Y |

OK CANCEL HELP

Figure 4. CPU Monitor STATUS dialog box

| | |
|---|---|
| Background color | -> (fig. 5b) |
| CPU graph color | -> (fig. 5a) |
| Freeze screen | Alt+F |
| Fill | Alt+I |

Figure 5. CPU Monitor OPTIONS dialog box

CPU 1
CPU 2
CPU 3
CPU 4
CPU 5
CPU 6
CPU 7
CPU 8

Figure 5a. CPU Monitor GRAPH Color CPU selection

NOTE: After selecting CPU, prompt for color selection (fig 5b).

WHITE
BLACK
BLUE
RED
PINK
GREEN
CYAN
YELLOW
DARK GRAY
DARK BLUE
DARK RED
DARK PINK
DARK GREEN
DARK CYAN
BROWN
PALE GRAY

Figure 5b. CPU Monitor Color selection

# Appendix A

The following is the source code for an actual PSD.

## Main program

```
#define INCL_ERROR_H

#include <os2.h>
#include <psd.h>
#include <alr.h>


extern ulong_t RMP2Available(void);


/*
 *  Global Variables
 */

P_F_2   router = 0;
char   *pParmString = 0;
int     IODelayCount = 30;
PLMA   *pPSDPLMA = 0;
ulong_t sizePLMA = 0;


/***  Disable - Disable interrupts
 *
 *    This function disables interrupts, and returns
 *    the original state of eflags
 *
 *    ENTRY   None
 *
 *    EXIT    EFLAGS
 *
 */

ulong_t Disable(void) {

   ulong_t eflags;

   _asm {
      pushfd
```

```
      pop   eax
      mov   eflags,eax
      cli
   };

   return (eflags);
}


/***  Enable - Restore the state of eflags
 *
 *    This function restores the state of eflags
 *
 *    ENTRY   eflags - state of eflags to restore
 *
 *    EXIT    None
 *
 */

void Enable(ulong_t eflags) {

   _asm {
      push  eflags
      popfd
   };

   return;
}


/***  InByte - Read a byte from a port
 *
 *    This function reads a byte from a specified port
 *
 *    ENTRY   port - port number to read from
 *
 *    EXIT    data read
 *
 */

ulong_t InByte(ulong_t port) {

   ulong_t data;
```

```
    _asm {
       mov   dx,port
       in    al,dx
       movzx eax,al
       mov   data,eax
    };

    return (data);
}


/***  OutByte - Writes a byte to a port
 *
 *    This function writes a byte to a specified port
 *
 *    ENTRY   port - port number to read from
 *            data - data to write
 *
 *    EXIT    None
 *
 */

void OutByte(ulong_t port, ulong_t data) {

   _asm {
      mov   dx,port
      mov   al,byte ptr data
      out   dx,al
   };

   return;
}


/***  SendEOI - Send an end of interrupt
 *
 *    This function sends an end of interrupt.
 *
 *    ENTRY   irq - irq level to end
 *
 *    EXIT    None
 *
 */
```

```
ulong_t SendEOI(ulong_t irq) {

   ulong_t flags;

   flags = Disable();

   if (irq < NUM_IRQ_PER_PIC)
      OutByte(PIC1_PORT0, OCW2_NON_SPECIFIC_EOI);
   else {
      OutByte(PIC2_PORT0, OCW2_NON_SPECIFIC_EOI);
      IODelay;
      OutByte(PIC1_PORT0, OCW2_NON_SPECIFIC_EOI);
   }

   Enable(flags);
}


/***  WHO_AM_I - Returns the current processor number
 *
 *    This function returns the current processor number
 *
 *    ENTRY   NONE
 *
 *    EXIT    Current processor number (P1 or P2)
 *
 */

ulong_t WHO_AM_I (void) {
   return(InByte(WHO_AM_I_PORT));
}


/***  IPIPresent - Detects the presence of an IPI
 *
 *    This function detects the presence of an IPI on the current
 *    processor
 *
 *    ENTRY   None
 *
 *    EXIT    NO_ERROR - IPI present
 *            -1       - IPI not present
```

```
 *
 */

ulong_t IPIPresent (void) {

   ulong_t rc = 0;
   struct control_s ctrl;
   ulong_t port;

   port = pPSDPLMA->controlport;

   ctrl.b_all = InByte(port);
   if (ctrl.b_387err)
   {
      OutByte (0xf0, 0); // The busy latch for NPX must be cleared.
                         // When we call the interrupt handler
                         // (w/ Call16bitDD int.asm), ints. are 1st enablec
                         // If the busy latch is not cleared, then we
                         // will take this interrupt in again and will
                         // eventually nest until the interrupt stack is
                         // overrun.
      rc = -1;
   }
   return (rc);
}


/***  Install - Install PSD
 *
 *    This function checks to see if this PSD is installable on the
 *    current platform.
 *
 *    ENTRY   pinstall - pointer to an INSTALL structure
 *
 *    EXIT    NO_ERROR - PSD Installed
 *            -1       - PSD not valid for this platform
 *
 */

ulong_t Install(INSTALL *pinstall) {

   VMALLOC vmac;
   int i;
   char *p;
```

```
    ulong_t rc = 0;
    char ALR_String =  "PROVEISA";

// _asm int 3;

    /* Setup Global variables */

    router = pinstall->pPSDHlpRouter;
    pParmString = pinstall->pParmString;
    pPSDPLMA = (void *)pinstall->pPSDPLMA;
    sizePLMA = pinstall->sizePLMA;

    vmac.addr = BIOS_SEG << 4;
    vmac.cbsize = _64K;
    vmac.flags = VMALLOC_PHYS;

    /* Map BIOS area */

    if ((rc = PSDHelp(router, PSDHLP_VMALLOC, &vmac)) == NO_ERROR) {

       /* Check for ALR string */

       p = (char *)vmac.addr + ALR_STRING_OFFSET;

       for (i = 0; ALR_String i != '\0'; i++)
          if (p i  != ALR_String i) {
             rc = -1;
             break;
          }

       /* Free BIOS mapping */
```

## Entry stub

```
.386

_TEXT SEGMENT

ASSUME CS:_TEXT,DS:NOTHING

      PUBLIC  _RMP2Available

_RMP2Available PROC

      mov   ah,0E2h
      mov   al,0
      int   15h
      movzx eax,ax
      retf

_RMP2Available ENDP

_TEXT ENDS

END
```

# PSD.H

```
/*static char *SCCSID = "@(#)psd.h 1.0 93/18/08";*/


// XLATOFF

#ifndef ulong_t

typedef unsigned long   ulong_t;
typedef unsigned short  ushort_t;
typedef unsigned char   uchar_t;

#endif

typedef int (*P_F_1)(ulong_t arg);
typedef int (*P_F_2)(ulong_t arg1, ulong_t arg2);

#define PSDHelp(router, function, arg) \
   ((*router)((function), (ulong_t)(arg)))

// XLATON
/* ASM
P_F_1 struc
dd ?
P_F_1 ends
P_F_2 struc
dd ?
P_F_2 ends
*/


#define WARM_REBOOT_VECTOR_SEG  0x40
#define WARM_REBOOT_VECTOR_OFF  0x67


/* PSD Info structure */

typedef struct info_s {                         /* psd */
   ulong_t  flags;                              /* PSD flags */
   ulong_t  version;                            /* PSD version */
```

```
   ulong_t  hmte;                               /* MTE handle of PSD */
   uchar_t *pParmString;                        /* Pointer to ASCIIZ PSD parame
   ulong_t  IRQ_IPI;                            /* IRQ for IPI */
   ulong_t  IRQ_LSI;                            /* IRQ for LSI */
   ulong_t  IRQ_SPI;                            /* IRQ for SPI */
} PSDINFO;


/* PSD flags definition */

#define PSD_ADV_INT_MODE          0x20000000  /* PSD is in adv int mode #815
#define PSD_INSTALLED             0x40000000  /* PSD has been installed */
#define PSD_INITIALIZED           0x80000000  /* PSD has been initialized */

/* PSD function numbers-structures */

#define PSD_INSTALL               0x00000000  /* Install PSD */

typedef struct install_s {                    /* install */
   P_F_2   pPSDHlpRouter;                     /* Address of PSDHlpRouter */
   char    *pParmString;                      /* Pointer to parameter string
   void    *pPSDPLMA;                         /* Pointer to PSD's PLMA */
   ulong_t sizePLMA;                          /* Size of PLMA in bytes */
} INSTALL;

#define PSD_DEINSTALL             0x00000001  /* DeInstall PSD */

#define PSD_INIT                  0x00000002  /* Initialize PSD */

typedef struct init_s {                       /* init */
   ulong_t flags;                             /* Init flags */
   ulong_t version;                           /* PSD Version number */
} INIT;

#define INIT_GLOBAL_IRQ_ACCESS    0x00000001  /* Platform has global IRQ acce
#define INIT_USE_FPERR_TRAP       0x00000002  /* Use Trap 16 to report FP er
#define INIT_EOI_IRQ13_ON_CPU0    0x00000004  /* eoi IRQ 13 only if on cpu 0
#define INIT_TIMER_CPU0           0x00000008  /* system timer is on CPU 0

#define PSD_PROC_INIT             0x00000003  /* Initialize processor */

#define PSD_START_PROC            0x00000004  /* Start processor */
```

```
#define PSD_GET_NUM_OF_PROCS    0x00000005  /* Get number of processors */

#define PSD_GEN_IPI             0x00000006  /* Generate an IPI */

#define PSD_END_IPI             0x00000007  /* End an IPI */

#define PSD_PORT_IO             0x00000008  /* Port I/O */

typedef struct port_io_s {                  /* port_io */
   ulong_t port;                            /* Port number to access */
   ulong_t data;                            /* Data read, or data to write
   ulong_t flags;                           /* IO Flags */
} PORT_IO;

#define IO_READ_BYTE    0x0000              /* Read a byte from the port */
#define IO_READ_WORD    0x0001              /* Read a word from the port */
#define IO_READ_DWORD   0x0002              /* Read a dword from the port *
#define IO_WRITE_BYTE   0x0003              /* Write a byte to the port */
#define IO_WRITE_WORD   0x0004              /* Write a word to the port */
#define IO_WRITE_DWORD  0x0005              /* Write a dword to the port */

#define IO_FLAGMASK     0x0007              /* Flag mask */

#define PSD_IRQ_MASK            0x00000009  /* Mask/Unmask IRQ levels */

typedef struct psd_irq_s {                  /* psd_irq */
   ulong_t flags;                           /* IRQ flags */
   ulong_t data;                            /* IRQ data */
                                            /*   depending on type of irq *
                                            /*   operation, the data field
                                            /*   can contain any of the */
                                            /*   following info: */
                                            /*   1) Mask or UNMasking data
                                            /*   2) IRR or ISR reg values *
                                            /*   3) IRQ # for EOI operatio
   ulong_t procnum;                         /* Processor number */
} PSD_IRQ;

#define PSD_IRQ_REG             0x0000000A  /* Access IRQ related regs */

#define PSD_IRQ_EOI             0x0000000B  /* Issue an EOI */

#define IRQ_MASK                0x00000001  /* Turn on IRQ mask bits */
#define IRQ_UNMASK              0x00000002  /* Turn off IRQ mask bits */
```

```
#define IRQ_GETMASK              0x00000004  /* Get IRQ mask bits */
#define IRQ_NEWMASK              0x00000010  /* Set and/or Reset all masks *
#define IRQ_READ_IRR             0x00000100  /* Read the IRR reg */
#define IRQ_READ_ISR             0x00000200  /* Read the ISR reg */

#define PSD_APP_COMM             0x0000000C  /* PSD/APP Communication */

#define PSD_SET_ADV_INT_MODE     0x0000000D  /* Set advanced int mode */

#define PSD_SET_PROC_STATE       0x0000000E  /* Set proc state; idle, or bus

#define PROC_STATE_IDLE          0x00000000  /* Processor is idle */
#define PROC_STATE_BUSY          0x00000001  /* Processor is busy */

#define PSD_QUERY_SYSTEM_TIMER   0x0000000F  /* Query Value of System Timer

typedef struct psd_qrytmr_s {                /* psd_qrytmr */
   ulong_t qw_ulLo_psd;                      /* Timer count */
   ulong_t qw_ulHi_psd;                      /* Timer count */
   ulong_t pqwTmr;                           /* 16:16 ptr to qwTmr */
} PSD_QRYTMR;

#define PSD_SET_SYSTEM_TIMER     0x00000010  /* Set System Timer 0 counter

typedef struct psd_settmr_s {                /* psd_settmr */
   ulong_t NewRollOver;                      /* NewRollover*/
   ulong_t pqwTmrRollover;                   /* 16:16 ptr to qwTmrRollover *
} PSD_SETTMR;

/* PSD helper function numbers-structures */

#define PSDHLP_VMALLOC           0x00000000  /* Allocate memory */

typedef struct vmalloc_s {                   /* vmalloc */
   ulong_t addr;                             /* Physical address to map */
                                             /*  if VMALLOC_PHYS */
                                             /* Lin addr to alloc at */
                                             /*  if VMALLOC_LOCSPECIFIC */
                                             /* on return, addr of allocatic
   ulong_t cbsize;                           /* Size of mapping in bytes */
   ulong_t flags;                            /* Allocation flags */
} VMALLOC;
```

```
#define VMALLOC_FIXED          0x00000001  /* Allocate resident memory */
#define VMALLOC_CONTIG         0x00000002  /* Allocate contiguous memory */
#define VMALLOC_LOCSPECIFIC    0x00000004  /* Alloc at a specific lin addr
#define VMALLOC_PHYS           0x00000008  /* Map physical address */
#define VMALLOC_1M             0x00000010  /* Allocate below 1M */

#define VMALLOC_FLAGMASK       0x0000001f  /* Valid flag mask */

#define PSDHLP_VMFREE          0x00000001  /* Free memory */

#define PSDHLP_SET_IRQ         0x00000002  /* Set up an IRQ */

typedef struct set_irq_s {                 /* set_irq */
   ushort_t irq;                           /* IRQ level */
   ushort_t flags;                         /* Set IRQ flags */
   ulong_t  vector;                        /* IRQ interrupt vector */
   P_F_2    handler;                       /* IRQ handler */
} SET_IRQ;

#define IRQf_IPI  0x0020                   /* IRQ for IPI */
#define IRQf_LSI  0x0040                   /* IRQ for LSI */
#define IRQf_SPI  0x0080                   /* IRQ for SPI */

#define PSDHLP_CALL_REAL_MODE  0x00000003  /* Call a function in real mode

typedef struct call_real_mode_s {          /* call_real_mode */
   ulong_t function;                       /* Function address */
   ulong_t pdata;                          /* Pointer to data area */
} CALL_REAL_MODE;

#define PSDHLP_VMLINTOPHYS     0x00000004  /* Convert linear addr to phys

#define PSDHLP_ADJ_PG_RANGES   0x00000005  /* Adjust page ranges */

typedef struct _pagerange_s {              /* pagerange */
   ulong_t lastframe;                      /* Last valid page in range */
   ulong_t firstframe;                     /* First valid page in range */
};

typedef struct adj_pg_ranges_s{            /* adj_pg_ranges */
   struct _pagerange_s *pprt;              /* Pointer to page range table
   ulong_t nranges;                        /* Num of ranges in range table
} ADJ_PG_RANGES;
```

```
/* PSD function prototypes */

extern void PSDEnter (ulong_t function, ulong_t arg, P_F_2 altEntry);
```

## Specific header

```
/*
 * Miscellaneous
 */

#define VERSION   0x00000010

#define _64K      (64 * 1024)

#define BIOS_SEG            0xF000
#define ALR_STRING_OFFSET   0xEC47

#define P2_AVAILABLE        0x00008000


/*
 * PLMA structure
 */

typedef struct plma_s {
   ulong_t procnum;       /* Current processor number (0-based) */
   ulong_t controlport;   /* Control port for current processor */
} PLMA;


/*
 * Generate delay between I/O instructions
 */

#define IODelay {int i; for(i = 0; i < IODelayCount; i++); }


/*
 * IPI info
 */

#define IPI_IRQ     0x0d      /* IRQ level for IPI */
#define IPI_VECTOR  0x75      /* Vector number for IPI */
```

```
/*
 * PIC Info
 */

#define NUM_IRQ_PER_PIC          0x08
#define OCW2_NON_SPECIFIC_EOI    0x20
#define PIC1_PORT0               0x20
#define PIC1_PORT1               0x21
#define PIC2_PORT0               0xA0
#define PIC2_PORT1               0xA1


/*
 * The contents of the WHO_AM_I port (read-only) can be used
 * by code to determine which processor we are currently on
 */

#define WHO_AM_I_PORT  0xC70
#define P1             0x00
#define P2             0xF0


/*
 * The processor control port contains the bits used to control
 * various functions of the associated processor
 */

#define P1_PROCESSOR_CONTROL_PORT  0x0C6A
#define P2_PROCESSOR_CONTROL_PORT  0xFC6A

struct _b_control_s {
   ulong_t _reset:1,      /* RESET - (Not implemented for P1) */
                          /*  1 = Resets processor */

           _387pres:1,    /* 387PRES - (Read only) */
                          /*  0 = 80387 is not installed */
                          /*  1 = 80387 is installed */

           _cacheon:1,    /* CACHEON - (Not implemented for P1) */
                          /*  0 = Disables cache */
                          /*  1 = Enables cache */
```

```
            _mbusaccess:1, /* M Bus Access (Not implemented for P1) */
                           /*  0 = Allows the processor to gain */
                           /*      control of the memory bus */
                           /*  1 = Prohibits the processor from gaining */
                           /*      access to the memory bus. The */
                           /*      processor can execute instructions */
                           /*      from its cache; however, cache read */
                           /*      misses, I/O, and writes cause the */
                           /*      processor to cease executing */
                           /*      instructions until the bit becomes */
                           /*      a "0" */

            _flush:1,      /* FLUSH */
                           /*  Writing a "1" to this bit followed by a "0" *
                           /*  causes invalidation of all cache address */
                           /*  information */

            _387err:1,     /* 387ERR */
                           /*  0 = No 80387 error */
                           /*  0 = An 80387 error has occurred. This bit */
                           /*      must be cleared by software */

            _pint:1,       /* PINT */
                           /*  A low-to-high transition of this bit causes *
                           /*  an interrupt. This bit must be cleared by  */
                           /*  software, preferably by the interrupt service
                           /*  routine. On P2, the value stored in FC68h */
                           /*  contains the interrupt number. P1 is always *
                           /*  interrupted with IRQ13 */

            _intdis:1,     /* INTDIS */
                           /*  When set to "1", this bit disables interrupts
                           /*  sent to the processor by way of the PINT bit.
                           /*  The PINT bit can still be changed when */
                           /*  interrupts are disabled; however, the */
                           /*  low-to-high transition is not seen by the */
                           /*  processor until the INTDIS bit is made inacti
            _pad:24;
};

struct _l_control_s {      /* to treat control as an unsigned long */
    unsigned long _long;
};
```

```
union _control_u {
    struct _b_control_s b_control_s;
    struct _l_control_s l_control_s;
};

struct control_s {
    union _control_u control_u;
};

#define b_reset       control_u.b_control_s._reset
#define b_387pres     control_u.b_control_s._387pres
#define b_cacheon     control_u.b_control_s._cacheon
#define b_mbusaccess  control_u.b_control_s._mbusaccess
#define b_flush       control_u.b_control_s._flush
#define b_387err      control_u.b_control_s._387err
#define b_pint        control_u.b_control_s._pint
#define b_intdis      control_u.b_control_s._intdis
#define b_all         control_u.l_control_s._long


/*
 * The interrupt vector control port contains the 8-bit interrupt
 * number that is executed when the PINT bit transitions from "0"
 * to "1". This vector is only used for P2. P1 is always interrupted
 * with IRQ 13.
 */

#define P2_INTERRUPT_VECTOR_CONTROL_PORT 0xFC68


/*
 * The following ports contain the EISA identification of the
 * system processor boards
 */

#define COMPAQ_ID1  0x0000000E
#define COMPAQ_ID2  0x00000011

#define P1_EISA_PRODUCT_ID_PORT1  0x0C80  /* Compressed COMPAQ ID - OEh */
#define P1_EISA_PRODUCT_ID_PORT2  0x0C81  /*                        11h */
#define P1_EISA_PRODUCT_ID_PORT3  0x0C82  /* Product code for the proc boa
#define P1_EISA_PRODUCT_ID_PORT4  0x0C83  /* Revision number */
```

```
#define P2_EISA_PRODUCT_ID_PORT1  0xFC80  /* Compressed COMPAQ ID - OEh */
#define P2_EISA_PRODUCT_ID_PORT2  0xFC81  /*                       11h */
#define P2_EISA_PRODUCT_ID_PORT3  0xFC82  /* Product code for the proc boai
#define P2_EISA_PRODUCT_ID_PORT4  0xFC83  /* Revision number */

/*
 * Any write to The RAM Relocation Register (memory mapped)
 * will flush the caches of both P1 and P2
 */

#define RAM_RELOCATION_REGISTER       0x80C00000


/*
 * The P1 Cache Control Register (memory mapped)
 */

#define P1_CACHE_CONTROL_REGISTER     0x80C00002

struct p1cache_s {
   ulong_t _reserved1:6,
           _p1cc:1,       /* P1 Cache Control */
                          /*  0 = Disables P1 cache */
                          /*  1 = Enables P1 cache */
           _reserved2:9;
};


/*
 * Expanision board control ports
 */

#define P1_EISA_EXPANSION_BOARD_CONTROL  0x0C84
#define P2_EISA_EXPANSION_BOARD_CONTROL  0xFC84
```

## Makefile

```
# SCCSID = @(#)makefile 6.7 92/06/03

#/***********************************************************************/
#/*                                                                     */
#/* PSD Name: ALR.PSD - ALR PSD                                         */
#/*          -----------------------------------                        */
#/*                                                                     */
#/* Source File Name: MAKEFILE                                          */
#/*                                                                     */
#/* Descriptive Name: MAKEFILE for the ALR PSD                          */
#/*                                                                     */
#/* Function:                                                           */
#/*                                                                     */
#/*                                                                     */
#/*---------------------------------------------------------------------*/
#/*                                                                     */
#/* Copyright (C) 1992 IBM Corporation                                  */
#/*                                                                     */
#/* DISCLAIMER OF WARRANTIES.  The following enclosed code is           */
#/* provided to you solely for the purpose of assisting you in          */
#/* the development of your applications. The code is provided          */
#/* "AS IS", without warranty of any kind. IBM shall not be liable      */
#/* for any damages arising out of your use of this code, even if       */
#/* they have been advised of the possibility of such damages.          */
#/*                                                                     */
#/*---------------------------------------------------------------------*/
#/*                                                                     */
#/* Change Log                                                          */
#/*                                                                     */
#/* Mark    Date      Programmer  Comment                               */
#/* ----    ----      ----------  -------                               */
#/* @nnnn   mm/dd/yy  NNN                                               */
#/*                                                                     */
#/*                                                                     */
#/***********************************************************************/

# ******  NOTE  ******
#
#       If you are using a SED command with TAB characters, many editors
```

```
#       will expand tabs causing unpredictable results in other programs.
#
#       Documentation:
#
#       Using SED command with TABS. Besure to invoke set tab save option
#       on your editor. If you don't, the program 'xyz' will not work
#       correctly.
#

#*************************************************************************
#  Dot directive definition area (usually just suffixes)
#*************************************************************************

.SUFFIXES:
.SUFFIXES: .com .sys .exe .obj .mbj .asm .inc .def .lnk .lrf .crf .ref
.SUFFIXES: .lst .sym .map .c .h .lib

#*************************************************************************
#  Environment Setup for the component(s).
#*************************************************************************

#
# Conditional Setup Area and User Defined Macros
#

#
# Compiler Location w/ includes, libs and tools
#

INC     = ..\..\..\inc
H       = ..\..\..\h
LIB     = ..\..\..\lib386;..\..\..\lib
TOOLSPATH = ..\..\..\tools

#
# Because the compiler/linker and other tools use environment
# variables ( INCLUDE, LIB, etc ) in order to get the location of files,
# the following line will check the environment for the LIFE of the
# makefile and will be specific to this set of instructions. All MAKEFILES
# are requested to use this format to insure that they are using the correc
# level of files and tools.
#
```

```
!if set INCLUDE=$(INC) || \
    set LIB=$(LIB) || set PATH=$(TOOLSPATH);$(DK_TOOLS)
!endif


#
# Compiler/tools Macros
#

AS=masm
CC=cl386
IMPLIB=implib
IPF=ipfc
LIBUTIL=lib
LINK=link386
MAPSYM=mapsym
RC=rc

#
# Compiler and Linker Options
#

AFLAGS = -MX -T -Z $(ENV)
AINC   = -I. -I$(INC)
CINC   = -I$(H) -I$(MAKEDIR)
CFLAGS = /c /Zp /Gs /AS $(ENV)
LFLAGS = /map /nod /exepack

LIBS = os2386.lib
DEF = ALR.def

#*************************************************************************
# Set up Macros that will contain all the different dependencies for the
# executables and dlls etc. that are generated.
#*************************************************************************

#
#
#
OBJ1 =  entry.obj main.obj

#
#       LIST Files
#
```

```
LIST =

OBJS = $(OBJ1)

#***************************************************************************
#   Setup the inference rules for compiling and assembling source code to
#   object code.
#***************************************************************************


.asm.obj:
        $(AS) $(AFLAGS) $(AINC) $*.asm;

.asm.mbj:
        $(AS) $(AFLAGS) -DMMIOPH $(AINC) $*.asm $*.mbj;

.asm.lst:
        $(AS) -l -n $(AFLAGS) $(AINC) $*.asm;

.c.obj:
        $(CC) $(CFLAGS) $(CINC) $*.c

.c.lst:
        $(CC) $(CFLAGS) /Fc $(CINC) $*.c
        copy $*.cod $*.lst
        del $*.cod


#***************************************************************************
#   Target Information
#***************************************************************************
#
# This is a very important step. The following small amount of code MUST
# NOT be removed from the program. The following directive will do
# dependency checking every time this component is built UNLESS the
# following is performed:
#               A specific tag is used -- ie. all
#
# This allows the developer as well as the B & I group to perform increment
# build with a degree of accuracy that has not been used before.
# There are some instances where certain types of INCLUDE files must be
# created first. This type of format will allow the developer to require
# that file to be created first. In order to achieve that, all that has to
```

```
# be done is to make the DEPEND.MAK tag have your required target. Below is
# an example:
#
#    depend.mak:   { your file(s) } dephold
#
# Please DON'T remove the following line
#

!include      "$(H)\common.mak"
!include      "$(H)\version.mak"

#
# Should be the default tag for all general processing
#

all:    ALR.psd

list: $(LIST)

clean:
        if exist *.lnk del *.lnk
        if exist *.obj del *.obj
        if exist *.mbj del *.mbj
        if exist *.map del *.map
        if exist *.old del *.old
        if exist *.lst del *.lst
        if exist *.lsd del *.lsd
        if exist *.sym del *.sym
        if exist *.sys del *.sys




#***************************************************************************
#   Specific Description Block Information
#***************************************************************************

# This section would only be for specific direction as to how to create
# unique elements that are necessary to the build process. This could
# be compiling or assembling, creation of DEF files and other unique
# files.
# If all compiler and assembly rules are the same, use an inference rule to
# perform the compilation.
#
```

```
alr.psd:  $(OBJS) makefile
        Rem Create DEF file <<$(DEF)
LIBRARY ALR

EXPORTS

   PSD_INSTALL           = _Install
   PSD_DEINSTALL         = _DeInstall
   PSD_INIT              = _Init
   PSD_PROC_INIT         = _ProcInit
   PSD_START_PROC        = _StartProcessor
   PSD_GET_NUM_OF_PROCS  = _GetNumOfProcs
   PSD_GEN_IPI           = _GenIPI
   PSD_END_IPI           = _EndIPI
<<keep
        $(LINK) $(LFLAGS) @<<$(@B).lnk
$(OBJ1)
$*.psd
$*.map
$(LIBS)
$(DEF)
<<keep
        $(MAPSYM) $*.map



#*************************************************************************
#  Dependency generation and Checking
#*************************************************************************

depend.mak:  dephold
```

# Glossary

**MP-unsafe**
: Does not provide the necessary serialization to run on more than one CPU at a time. For example, a driver will be MP unsafe if it relies upon priorities between threads for accessing shared resources, or uses the CLI/STI method for protecting resources like semaphores or memory.

**MP-safe**
: Provides the necessary serialization to run properly in a system with greater than one processor. Does not use invalid UP serialization techniques. For example, a driver will be MP safe if it *does not* rely upon priorities between threads for accessing shared resources, or use the CLI/STI method for protecting resources like semaphores or memory.

**MP-exploitive**
: Provides proper MP serialization techniques which allow multiple threads to run concurrently on more than one CPU.